Package ‘deSolve’

May 9, 2018

Version 1.21

Title Solvers for Initial Value Problems of Differential Equations ('ODE', 'DAE', 'DDE')

Author Karline Soetaert [aut],
Thomas Petzoldt [aut, cre],
R. Woodrow Setzer [aut],
odepack authors [cph]

Maintainer Thomas Petzoldt <thomas.petzoldt@tu-dresden.de>

Depends R (>= 2.15.0)

Imports methods, graphics, grDevices, stats

Suggests scatterplot3d

Description Functions that solve initial value problems of a system
of first-order ordinary differential equations ('ODE'), of
partial differential equations ('PDE'), of differential
algebraic equations ('DAE'), and of delay differential
equations. The functions provide an interface to the FORTRAN
functions 'lsoda', 'lsodar', 'lsode', 'lsodes' of the 'ODEPACK'
collection, to the FORTRAN functions 'dvode', 'zvode' and 'daspk'
and a C-implementation of solvers of the 'Runge-Kutta' family with
fixed or variable time steps. The package contains routines
designed for solving 'ODEs' resulting from 1-D, 2-D and 3-D
partial differential equations ('PDE') that have been converted
to 'ODEs' by numerical differencing.

License GPL (>= 2)

URL http://desolve.r-forge.r-project.org/

LazyData yes

NeedsCompilation yes

Repository CRAN

Date/Publication 2018-05-09 09:38:07 UTC
deSolve-package

R topics documented:

deSolve-package .................................................. 2
aquaphy .......................................................... 5
ccl4data .......................................................... 8
ccl4model ........................................................ 9
cleanEventTimes .............................................. 12
daspk ............................................................. 13
dede ............................................................... 23
diagnostics ...................................................... 27
diagnostics.deSolve ........................................... 28
DLLfunc ........................................................... 29
DLLres ............................................................ 31
events .............................................................. 33
forcings ............................................................ 39
lsoda ............................................................... 42
lsodar ............................................................. 49
lsode .............................................................. 55
lsodes ............................................................. 63
ode ................................................................. 71
ode.1D ............................................................ 77
ode.2D ............................................................ 83
ode.3D ............................................................ 89
ode.band ........................................................ 92
plot.deSolve .................................................... 95
radau ............................................................... 102
rk ................................................................. 109
rk4 ................................................................. 115
rkMethod ........................................................ 118
SCOC ............................................................ 123
timelags ........................................................... 125
vode ............................................................... 128
zvode ............................................................. 135

Index 141


Description

Functions that solve initial value problems of a system of first-order ordinary differential equations (ODE), of partial differential equations (PDE), of differential algebraic equations (DAE) and delay differential equations.
The functions provide an interface to the FORTRAN functions lsoda, lsodar, lsode, lsodes of the ODEPACK collection, to the FORTRAN functions dvode and daspk and a C-implementation of solvers of the Runge-Kutta family with fixed or variable time steps.

The package contains routines designed for solving ODEs resulting from 1-D, 2-D and 3-D partial differential equations (PDE) that have been converted to ODEs by numerical differencing. It includes root-finding (or event location) and provides access to lagged variables and derivatives.

Details

<table>
<thead>
<tr>
<th>Package</th>
<th>deSolve</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Package</td>
</tr>
<tr>
<td>Version</td>
<td>1.21</td>
</tr>
<tr>
<td>Date</td>
<td>2018-05-08</td>
</tr>
<tr>
<td>License</td>
<td>GNU Public License 2 or above</td>
</tr>
</tbody>
</table>

The system of differential equations is written as an R function or defined in compiled code that has been dynamically loaded, see package vignette compiledCode for details. The solvers may be used as part of a modeling package for differential equations, or for parameter estimation using any appropriate modeling tool for non-linear models in R such as optim, nls, nlm or nlme, or FME.

Package Vignettes, Examples, Online Resources

- Solving Initial Value Differential Equations in R (pdf, R code)
- Writing Code in Compiled Languages (pdf, R code)
- Examples in R (./.doc/examples), and in Fortran or C (./.doc/dynload, ./.doc/dynload-dede)
- deSolve homepage: http://desolve.r-forge.r-project.org (Papers, Books, PDFs)
- Mailing list: mailto:r-sig-dynamic-models@r-project.org

Author(s)

Karline Soetaert, Thomas Petzoldt, R. Woodrow Setzer

References


See also the references given on the specific help pages of the different methods.

See Also
ode for a general interface to most of the ODE solvers,
ode.band for solving models with a banded Jacobian,
ode.1D, ode.2D, ode.3D, for integrating 1-D, 2-D and 3-D models,
dede for a general interface to the delay differential equation solvers,
lsoda, lsode, lsodes, lsodar, vode, for ODE solvers of the Livermore family,
daspk, for a DAE solver up to index 1, of the Livermore family,
radau for integrating DAEs up to index 3 using an implicit Runge-Kutta,
rk, rkMethod, rk4, euler for Runge-Kutta solvers,
DLLfunc, DLLres, for testing model implementations in compiled code,
forcings, events, for how to implement forcing functions (external variables) and events (sudden changes in state variables),
lagvalue, lagderiv, for how to get access to lagged values of state variables and derivatives.

Examples
library(deSolve)

## Chaos in the atmosphere
Lorenz <- function(t, state, parameters) {
  with(as.list(c(state, parameters)), {
    dx <- a * X + Y * Z
    dy <- b * (Y - Z)
    dz <- -X * Y + c * Y - Z
    list(c(dx, dy, dz))
  })
}

parameters <- c(a = -8/3, b = -10, c = 28)
state <- c(X = 1, Y = 1, Z = 1)
times <- seq(0, 100, by = 0.01)

out <- ode(y = state, times = times, func = Lorenz, parms = parameters)

plot(out)

## add a 3D figure if package scatterplot3D is available
if (require(scatterplot3D))
  scatterplot3D(out[, -1], type = "l")
Description

A phytoplankton model with uncoupled carbon and nitrogen assimilation as a function of light and Dissolved Inorganic Nitrogen (DIN) concentration.

Algal biomass is described via 3 different state variables:

- low molecular weight carbohydrates (LMW), the product of photosynthesis,
- storage molecules (RESERVE) and
- the biosynthetic and photosynthetic apparatus (PROTEINS).

All algal state variables are expressed in mmol C m$^{-3}$. Only proteins contain nitrogen and chlorophyll, with a fixed stoichiometric ratio. As the relative amount of proteins changes in the algae, so does the N:C and the Chl:C ratio.

An additional state variable, dissolved inorganic nitrogen (DIN) has units of mmol N m$^{-3}$.

The algae grow in a dilution culture (chemostat): there is constant inflow of DIN and outflow of culture water, including DIN and algae, at the same rate.

Two versions of the model are included.

- In the default model, there is a day-night illumination regime, i.e. the light is switched on and off at fixed times (where the sum of illuminated + dark period = 24 hours).
- In another version, the light is imposed as a forcing function data set.

This model is written in FORTRAN.

Usage

aquaphy(times, y, parms, PAR = NULL, ...)

Arguments

times  time sequence for which output is wanted; the first value of times must be the initial time,
y      the initial (state) values ("DIN", "PROTEIN", "RESERVE", "LMW"), in that order,
parms  vector or list with the aquaphy model parameters; see the example for the order in which these have to be defined.
PAR    a data set of the photosynthetically active radiation (light intensity), if NULL, on-off PAR is used,
...    any other parameters passed to the integrator ode (which solves the model).
Details

The model is implemented primarily to demonstrate the linking of FORTRAN with R-code. The source can be found in the ‘doc/examples/dynload’ subdirectory of the package.

Author(s)

Karline Soetaert <karline.soetaert@nioz.nl>

References


See Also

ccl4model, the CC14 inhalation model.

Examples

```r
parameters <- c(
  maxPhotoSynt = 0.125,  # mol C/mol C/hr
  rMortPHY = 0.001,     # /hr
  alpha = -0.125/150,   # uEin/st/m2/s/hr
  pExudation = 0.0,      # -
  maxProteinSynt = 1.36, # mol C/mol C/hr
  ksDIN = 1.0,           # mmol N/m3
  minPLMW = 0.05,        # mol C/mol C
  maxPLMW = 0.15,        # mol C/mol C
  minQuotum = 0.075,     # mol C/mol C
  maxStorage = 0.23,     # /h
  respirationRate= 0.0001,  # /h
  pResp = 0.4,          # -
  catabolismRate = 0.06, # /h
  dilutionRate = 0.01,   # /h
  rNCPProtein = 0.2,     # mol N/mol C
  inputDIN = 10.0,       # mmol N/m3
  rChlN = 1,            # g Chl/mol N
  parMean = 250.0)      # umol Phot/m2/s
```

dayLength = 15. # hours
#
## The initial conditions
## ------------------------
state <- c(DIN = 6., # mmol N/m3
    PROTEIN = 20.0, # mmol C/m3
    RESERVE = 5.0, # mmol C/m3
    LMW = 1.0) # mmol C/m3
#
## Running the model
## ------------------------
times <- seq(0, 24*20, 1)
out <- as.data.frame(aquaphy(times, state, parameters))
#
## Plotting model output
## ------------------------
par(mfrow = c(2, 2), oma = c(0, 0, 3, 0))
col <- grey(0.9)
ii <- 1:length(out$PAR)

plot(times[ii], out$Chlorophyll[ii], type = "l",
    main = "Chlorophyll", xlab = "time, hours", ylab = "ug/l")
polygon(times[ii], out$PAR[ii]-10, col = col, border = NA); box()
lines(times[ii], out$Chlorophyll[ii], lwd = 2 )

plot (times[ii], out$DIN[ii], type = "l", main = "DIN",
    xlab = "time, hours",ylab = "mmolN/m3")
polygon(times[ii], out$PAR[ii]-10, col = col, border = NA); box()
lines(times[ii], out$DIN[ii], lwd = 2 )

plot (times[ii], out$NCratio[ii], type = "n", main = "NCratio",
    xlab = "time, hours", ylab = "molN/molC")
polygon(times[ii], out$PAR[ii]-10, col = col, border = NA); box()
lines(times[ii], out$NCratio[ii], lwd = 2 )

plot (times[ii], out$Photosynthesis[ii].type = "l",
    main = "Photosynthesis", xlab = "time, hours",
    ylab = "mmolC/m3/hr")
polygon(times[ii], out$PAR[ii]-10, col = col, border = NA); box()
lines(times[ii], out$Photosynthesis[ii], lwd = 2 )
mtext(outer = TRUE, side = 3, "AQUAPHY, PAR= on-off", cex = 1.5)
## Description

The results of a closed chamber experiment to determine metabolic parameters for CCl4 (carbon tetrachloride) in rats.
Usage
data(ccl4data)

Format
This data frame contains the following columns:

- **time**: the time (in hours after starting the experiment).
- **initconc**: initial chamber concentration (ppm).
- **animal**: this is a repeated measures design; this variable indicates which animal the observation pertains to.
- **ChamberConc**: chamber concentration at time, in ppm.

Source

Examples
plot(ChamberConc ~ time, data = ccl4data, xlab = "Time (hours)",
     xlim = range(c(0, ccl4data$time)),
     ylab = "Chamber Concentration (ppm)", log = "y")
ccl4data.avg <- aggregate(ccl4data$ChamberConc,
                           by = ccl4data[c("time", "initconc")], mean)
points(x ~ time, data = ccl4data.avg, pch = 16)

THE CCL4 INHALATION MODEL

Description
The CCl4 inhalation model implemented in Fortran

Usage
```
ccl4model(times, y, parms, ...)
```

Arguments

- **times**: time sequence for which the model has to be integrated.
- **y**: the initial values for the state variables ("AI", "AAM", "AT", "AF", "AL", "CLT" and "AM"), in that order.
- **parms**: vector or list holding the ccl4 model parameters; see the example for the order in which these have to be defined.
- **...**: any other parameters passed to the integrator ode (which solves the model).
Details

The model is implemented primarily to demonstrate the linking of FORTRAN with R-code. The source can be found in the ‘/doc/examples/dynload’ subdirectory of the package.

Author(s)

R. Woodrow Setzer <setzer.woodrow@epa.gov>

See Also

Try `demo(ccl4model)` for how this model has been fitted to the dataset `ccl4data`, `aquaphy`, another FORTRAN model, describing growth in aquatic phytoplankton.

Examples

```r
## Parameter values

Pm <- c(
  # Physiological parameters
  BW = 0.182, # Body weight (kg)
  QP = 4.0,  # Alveolar ventilation rate (hr^-1)
  QC = 4.0,  # Cardiac output (hr^-1)
  VFC = 0.08, # Fraction fat tissue (kg/(kg/BW))
  VLC = 0.04, # Fraction liver tissue (kg/(kg/BW))
  VMC = 0.74, # Fraction of muscle tissue (kg/(kg/BW))
  QFC = 0.05, # Fractional blood flow to fat ((hr^-1)/QC)
  QLC = 0.15, # Fractional blood flow to liver ((hr^-1)/QC)
  QMC = 0.32, # Fractional blood flow to muscle ((hr^-1)/QC)

  # Chemical specific parameters for chemical
  PLA = 16.17, # Liver/air partition coefficient
  PFA = 281.48, # Fat/air partition coefficient
  PMA = 13.3, # Muscle/air partition coefficient
  PTA = 16.17, # Viscera/air partition coefficient
  PB = 5.487,  # Blood/air partition coefficient
  MW = 153.8,  # Molecular weight (g/mol)
  VMAX = 0.04321671, # Max. velocity of metabolism (mg/hr) -calibrated
  KM = 0.4027255, # Michaelis-Menten constant (mg/l) -calibrated

  # Parameters for simulated experiment
  CONC = 1000, # Inhaled concentration
  KL = 0.02,  # Loss rate from empty chamber /hr
  RATS = 1.0,  # Number of rats enclosed in chamber
  VCHC = 3.8  # Volume of closed chamber (l)
)
```

## State variables

```r
```
```r
y <- c(
  AI = 21, # total mass, mg
  AAM = 0,
  AT = 0,
  AF = 0,
  AL = 0,
  CLT = 0, # area under the conc.-time curve in the liver
  AM = 0 # the amount metabolized (AM)
)

## Model application

times <- seq(0, 6, by = 0.1)

## initial inhaled concentration-calibrated
conc <- c(26.496, 90.197, 245.15, 951.46)

plot(ChamberConc ~ time, data = ccl4data, xlab = "Time (hours)",
     ylim = range(c(0, ccl4data$time)),
     ylab = "Chamber Concentration (ppm)",
     log = "y", main = "ccl4model")

for (cc in conc) {
  Pm["CONC"] <- cc

  VCH <- Pm["VCHC"] - Pm["RATS"] * Pm["BW"]
  AI0 <- VCH * Pm["CONC"] * Pm["MW"]/24450
  y["AI"] <- AI0

  ## run the model:
  out <- as.data.frame(ccl4model(times, y, Pm))
  lines(out$time, out$CP, lwd = 2)
}

legend("topright", lty = c(NA, 1), pch = c(1, NA), lwd = c(NA, 2),
       legend = c("data", "model"))

## An example with tracer injection

## every day, a conc of 2 is added to AI.
# 1. implemented as a data.frame
eventdat <- data.frame(var = rep("AI", 6), time = 1:6, value = rep(1, 6), method = rep("add", 6))

print(system.time(
  out <- ccl4model(times, y, Pm, events = list(data = eventdat))
))
```
plot(out, mfrow = c(3, 4), type = "l", lwd = 2)

# 2. implemented as a function in a DLL!
print(system.time(
  out2 <- ccl4model(times, y, Pm, events = list(func = "eventfun", time = 1:6))
))
plot(out2, mfrow = c(3, 4), type = "l", lwd = 2)

---

**cleanEventTimes**

*Find Nearest Event for Each Time Step and Clean Time Steps to Avoid Doubles*

**Description**

These functions can be used for checking time steps and events used by ode solver functions. They are normally called internally within the solvers.

**Usage**

```r
nearestEvent(times, eventtimes)
cleanEventTimes(times, eventtimes, eps = .Machine$double.eps * 10)
```

**Arguments**

- `times` the vector of output times,
- `eventtimes` a vector with the event times,
- `eps` relative tolerance value below which two numbers are assumed to be numerically equal.

**Details**

In floating point arithmetics, problems can occur if values have to be compared for 'equality' but are only close to each other and not exactly the same.

The utility functions can be used to add all `eventtimes` to the output `times` vector, but without including times that are very close to an event.

This means that all values of `eventtimes` are contained but only the subset of times that have no close neighbors in `eventtimes`.

These checks are normally performed internally by the integration solvers.

**Value**

- `nearestEvent` returns a vector with the closest events for each time step and
- `cleanEventTimes` returns a vector with the output times without all those that are 'very close' to an event.
**daspk**

**Author(s)**

Thomas Petzoldt

**See Also**

events

**Examples**

```r
events <- sort(c(0, 2, 3, 4 + 1e-10, 5, 7 - 1e-10, 7 + 6e-15, 7.5, 9, 24.9999, 25, 80, 1001, 1e300))
times <- sort(c(0, 1:7, 4.5, 6.75, 7.5, 9.2, 9.0001, 25, 879, 1e3, 1e300+5))

nearest <- nearestEvent(times, events)
data.frame(times=times, nearest = nearest)

## typical usage: include all events in times after removing values that
## are numerically close together, events have priority
times
unique_times <- cleanEventTimes(times, events)
newtimes <- sort(c(unique_times, events))
newtimes
```

---

**daspk**

*Solver for Differential Algebraic Equations (DAE)*

**Description**

Solves either:

- a system of ordinary differential equations (ODE) of the form
  \[ y' = f(t, y,...) \]
  or
- a system of differential algebraic equations (DAE) of the form
  \[ F(t, y, y') = 0 \]
  or
- a system of linearly implicit DAEs in the form
  \[ My' = f(t, y) \]

using a combination of backward differentiation formula (BDF) and a direct linear system solution method (dense or banded).

The R function `daspk` provides an interface to the FORTRAN DAE solver of the same name, written by Linda R. Petzold, Peter N. Brown, Alan C. Hindmarsh and Clement W. Ulrich.

The system of DE's is written as an R function (which may, of course, use `.C`, `.Fortran`, `.Call`, etc., to call foreign code) or be defined in compiled code that has been dynamically loaded.
Usage

daspk(y, times, func = NULL, parms, nind = c(length(y), 0, 0),
  dy = NULL, res = NULL, nalg = 0,
  rtol = 1e-6, atol = 1e-6, jacfunc = NULL,
  jacres = NULL, jactype = "fullint", mass = NULL, estini = NULL,
  verbose = FALSE, tcrit = NULL, hmin = 0, hmax = NULL,
  hini = 0, ynames = TRUE, maxord = 5, bandup = NULL,
  banddown = NULL, maxsteps = 5000, dllname = NULL,
  initfunc = dllname, initpar = parms, rpar = NULL,
  ipar = NULL, nout = 0, outnames = NULL,
  forcings=NULL, initforc = NULL, fcontrol=0,
  events = NULL, lags = NULL, ...)

Arguments

y the initial (state) values for the DE system. If y has a name attribute, the
names will be used to label the output matrix.

times time sequence for which output is wanted; the first value of
	times must be the initial time; if only one step is to be taken; set
	times = NULL.

func to be used if the model is an ODE, or a DAE written in linearly implicit form
(M y' = f(t, y)). func should be an R-function that computes the values of the
	derivatives in the ODE system (the model definition) at time t.
	func must be defined as: func <- function(t, y, parms,...).

t is the current time point in the integration, y is the current estimate of the
	variables in the ODE system. If the initial values y has a names attribute, the
names will be available inside func, unless ynames is FALSE. parms is a vector
	or list of parameters. ... (optional) are any other arguments passed to the
function.

The return value of func should be a list, whose first element is a vector con-
taining the derivatives of y with respect to time, and whose next elements are

global values that are required at each point in times. The derivatives should be
specified in the same order as the specification of the state variables y.

Note that it is not possible to define func as a compiled function in a dynamically
loaded shared library. Use res instead.

parms vector or list of parameters used in func, jacfunc, or res

nind if a DAE system: a three-valued vector with the number of variables of index 1,
	2, 3 respectively. The equations must be defined such that the index 1 variables
precede the index 2 variables which in turn precede the index 3 variables. The
sum of the variables of different index should equal N, the total number of vari-
ables. Note that this has been added for consistency with radau. If used, then
the variables are weighed differently than in the original daspk code, i.e. index
2 variables are scaled with 1/h, index 3 variables are scaled with 1/h^2. In some

cases this allows daspk to solve index 2 or index 3 problems.

dy the initial derivatives of the state variables of the DE system. Ignored if an ODE.

res if a DAE system: either an R-function that computes the residual function
	F(t, y, y') of the DAE system (the model definition) at time t, or a character


string giving the name of a compiled function in a dynamically loaded shared library.

If res is a user-supplied R-function, it must be defined as: res <- function(t, y, dy, parms, ...).
Here t is the current time point in the integration, y is the current estimate of the variables in the ODE system, dy are the corresponding derivatives. If the initial y or dy have a names attribute, the names will be available inside res, unless ynames is FALSE. parms is a vector of parameters.

The return value of res should be a list, whose first element is a vector containing the residuals of the DAE system, i.e. \( \delta = F(t, y, y') \), and whose next elements contain output variables that are required at each point in times.

If res is a string, then dllname must give the name of the shared library (without extension) which must be loaded before daspk() is called (see package vignette "compiledCode" for more information).

**nalg**
if a DAE system: the number of algebraic equations (equations not involving derivatives). Algebraic equations should always be the last, i.e. preceded by the differential equations.

Only used if estini = 1.

**rtol**
relative error tolerance, either a scalar or a vector, one value for each y.

**atol**
absolute error tolerance, either a scalar or a vector, one value for each y.

**jacfunc**
if not NULL, an R function that computes the Jacobian of the system of differential equations. Only used in case the system is an ODE \( y' = f(t, y) \), specified by func. The R calling sequence for jacfunc is identical to that of func.

If the Jacobian is a full matrix, jacfunc should return a matrix \( \frac{\partial y}{\partial y} \), where the ith row contains the derivative of \( dy_i/dt \) with respect to \( y_j \), or a vector containing the matrix elements by columns (the way R and FORTRAN store matrices).

If the Jacobian is banded, jacfunc should return a matrix containing only the nonzero bands of the Jacobian, rotated row-wise. See first example of lsode.

**jacres**
jacres and not jacfunc should be used if the system is specified by the residual function \( F(t, y, y') \), i.e. jacres is used in conjunction with res.

If jacres is an R-function, the calling sequence for jacres is identical to that of res, but with extra parameter cj. Thus it should be called as: jacres = func(t, y, dy, parms, cj, ...). Here t is the current time point in the integration, y is the current estimate of the variables in the ODE system, y' are the corresponding derivatives and cj is a scalar, which is normally proportional to the inverse of the stepsize. If the initial y or dy have a names attribute, the names will be available inside jacres, unless ynames is FALSE. parms is a vector of parameters (which may have a names attribute).

If the Jacobian is a full matrix, jacres should return the matrix \( dG/dy + cj \cdot dG/dy' \), where the ith row is the sum of the derivatives of \( G_i \) with respect to \( y_j \) and the scaled derivatives of \( G_i \) with respect to \( y'_j \).

If the Jacobian is banded, jacres should return only the nonzero bands of the Jacobian, rotated row-wise. See details for the calling sequence when jacres is a string.

**jactype**
the structure of the Jacobian, one of "fullint", "fullusr", "bandusr" or "bandint" - either full or banded and estimated internally or by the user.
the mass matrix. If not NULL, the problem is a linearly implicit DAE and defined as $M \frac{dy}{dt} = f(t, y)$. The mass-matrix $M$ should be of dimension $n \times n$ where $n$ is the number of $y$-values.

If mass=NULL then the model is either an ODE or a DAE, specified with res

estini only if a DAE system, and if initial values of $y$ and $dy$ are not consistent (i.e. $F(t, y, dy) \neq 0$), setting estini = 1 or 2, will solve for them. If estini = 1: $dy$ and the algebraic variables are estimated from $y$; in this case, the number of algebraic equations must be given (nalg). If estini = 2: $y$ will be estimated from $dy$.

verbose if TRUE: full output to the screen, e.g. will print the diagnostics of the integration - see details.

tcrit the FORTRAN routine daspk overshoots its targets (times points in the vector times), and interpolates values for the desired time points. If there is a time beyond which integration should not proceed (perhaps because of a singularity), that should be provided in tcrit.

hmin an optional minimum value of the integration stepsize. In special situations this parameter may speed up computations with the cost of precision. Don’t use hmin if you don’t know why!

hmax an optional maximum value of the integration stepsize. If not specified, hmax is set to the largest difference in times, to avoid that the simulation possibly ignores short-term events. If 0, no maximal size is specified.

hini initial step size to be attempted; if 0, the initial step size is determined by the solver

ynames logical, if FALSE, names of state variables are not passed to function func; this may speed up the simulation especially for large models.

maxord the maximum order to be allowed. Reduce maxord to save storage space ( <= 5)

bandup number of non-zero bands above the diagonal, in case the Jacobian is banded (and jactype one of "bandint", "bandusr")

banddown number of non-zero bands below the diagonal, in case the Jacobian is banded (and jactype one of "bandint", "bandusr")

maxsteps maximal number of steps per output interval taken by the solver; will be recalculated to be at least 500 and a multiple of 500; if verbose is TRUE the solver will give a warning if more than 500 steps are taken, but it will continue till maxsteps steps. (Note this warning was always given in deSolve versions < 1.10.3).

dllname a string giving the name of the shared library (without extension) that contains all the compiled function or subroutine definitions referred to in res and jacres. See package vignette "compiledCode".

initfunc if not NULL, the name of the initialisation function (which initialises values of parameters), as provided in ‘dllname’. See package vignette "compiledCode".

initpar only when ‘dllname’ is specified and an initialisation function initfunc is in the dll: the parameters passed to the initialiser, to initialise the common blocks (FORTRAN) or global variables (C, C++).
**rpar**  
only when ‘dllname’ is specified: a vector with double precision values passed to the dll-functions whose names are specified by `res` and `jacres`.

**ipar**  
only when ‘dllname’ is specified: a vector with integer values passed to the dll-functions whose names are specified by `res` and `jacres`.

**nout**  
only used if ‘dllname’ is specified and the model is defined in compiled code: the number of output variables calculated in the compiled function `res`, present in the shared library. Note: it is not automatically checked whether this is indeed the number of output variables calculated in the dll - you have to perform this check in the code - See package vignette "compiledCode".

**outnames**  
only used if ‘dllname’ is specified and nout > 0: the names of output variables calculated in the compiled function `res`, present in the shared library. These names will be used to label the output matrix.

**forcings**  
only used if ‘dllname’ is specified: a list with the forcing function data sets, each present as a two-columned matrix, with (time,value); interpolation outside the interval [min(times), max(times)] is done by taking the value at the closest data extreme.  
See forcings or package vignette "compiledCode".

**initforc**  
if not NULL, the name of the forcing function initialisation function, as provided in ‘dllname’. It MUST be present if forcings has been given a value. See forcings or package vignette "compiledCode".

**fcontrol**  
A list of control parameters for the forcing functions. See forcings or vignette compiledCode.

**events**  
A matrix or data frame that specifies events, i.e. when the value of a state variable is suddenly changed. See events for more information.

**lags**  
A list that specifies timelags, i.e. the number of steps that has to be kept. To be used for delay differential equations. See timelags, dede for more information.

...  
additional arguments passed to `func`, `jacfunc`, `res` and `jacres`, allowing this to be a generic function.

**Details**

The daspk solver uses the backward differentiation formulas of orders one through five (specified with `maxord`) to solve either:

- an ODE system of the form
  \[ y' = f(t, y, ...) \]
  or

- a DAE system of the form
  \[ y' = M f(t, y, ...) \]
  or

- a DAE system of the form
  \[ F(t, y, y') = 0 \]

. The index of the DAE should be preferable <= 1.
ODEs are specified using argument `func`, DAEs are specified using argument `res`.

If a DAE system, Values for $y$ and $y'$ (argument `dy`) at the initial time must be given as input. Ideally, these values should be consistent, that is, if $t$, $y$, $y'$ are the given initial values, they should satisfy $F(t,y,y') = 0$.

However, if consistent values are not known, in many cases `daspk` can solve for them: when `estini` = 1, $y'$ and algebraic variables (their number specified with `nalg`) will be estimated, when `estini` = 2, $y$ will be estimated.

The form of the Jacobian can be specified by `jactype`. This is one of:

- `jactype = "fullint"`: a full Jacobian, calculated internally by `daspk`, the default,
- `jactype = "fullusr"`: a full Jacobian, specified by user function `jacfunc` or `jacres`,
- `jactype = "bandusr"`: a banded Jacobian, specified by user function `jacfunc` or `jacres`; the size of the bands specified by `bandup` and `banddown`,
- `jactype = "bandint"`: a banded Jacobian, calculated by `daspk`; the size of the bands specified by `bandup` and `banddown`.

If `jactype = "fullusr"` or "bandusr" then the user must supply a subroutine `jacfunc`.

If `jactype = "fullusr"` or "bandusr" then the user must supply a subroutine `jacfunc` or `jacres`.

The input parameters `rtol`, and `atol` determine the error control performed by the solver. If the request for precision exceeds the capabilities of the machine, `daspk` will return an error code. See `lsoda` for details.

When the index of the variables is specified (argument `nind`), and higher index variables are present, then the equations are scaled such that equations corresponding to index 2 variables are multiplied with $1/h$, for index 3 they are multiplied with $1/h^2$, where $h$ is the time step. This is not in the standard DASPK code, but has been added for consistency with solver `radau`. Because of this, `daspk` can solve certain index 2 or index 3 problems.

`res` and `jacres` may be defined in compiled C or FORTRAN code, as well as in an R-function. See package vignette "compiledCode" for details. Examples in FORTRAN are in the `dynload` subdirectory of the `desolve` package directory.

The diagnostics of the integration can be printed to screen by calling `diagnostics`. If `verbose = TRUE`, the diagnostics will written to the screen at the end of the integration.

See vignette("deSolve") for an explanation of each element in the vectors containing the diagnostic properties and how to directly access them.

Models may be defined in compiled C or FORTRAN code, as well as in an R-function. See package vignette "compiledCode" for details.

More information about models defined in compiled code is in the package vignette ("compiledCode"); information about linking forcing functions to compiled code is in `forcings`.

Examples in both C and FORTRAN are in the `dynload` subdirectory of the `desolve` package directory.

Value

A matrix of class `deSolve` with up to as many rows as elements in `times` and as many columns as elements in `y` plus the number of "global" values returned in the next elements of the return from `func` or `res`, plus an additional column (the first) for the time value. There will be one row for each
element in times unless the FORTRAN routine ‘daspk’ returns with an unrecoverable error. If y has a names attribute, it will be used to label the columns of the output value.

**Note**

In this version, the Krylov method is not (yet) supported.

From deSolve version 1.10.4 and above, the following changes were made:

1. the argument list to daspk now also includes nind, the index of each variable. This is used to scale the variables, such that daspk in R can also solve certain index 2 or index 3 problems, which the original Fortran version may not be able to solve.
2. the default of atol was changed from 1e-8 to 1e-6, to be consistent with the other solvers.
3. the multiple warnings from daspk when the number of steps exceed 500 were toggled off unless verbose is TRUE

**Author(s)**

Karline Soetaert <karline.soetaert@nioz.nl>

**References**


Netlib: [http://www.netlib.org](http://www.netlib.org)

**See Also**

- radau for integrating DAEs up to index 3,
- rk,
- rk4 and euler for Runge-Kutta integrators.
- lsoda, lsode, lsodes, lsodar, vode, for other solvers of the Livermore family,
- ode for a general interface to most of the ODE solvers,
- ode.band for solving models with a banded Jacobian,
- ode.1D for integrating 1-D models,
- ode.2D for integrating 2-D models,
- ode.3D for integrating 3-D models,
- diagnostics to print diagnostic messages.
Examples

```r
## Coupled chemical reactions including an equilibrium
## modeled as (1) an ODE and (2) as a DAE
##
## The model describes three chemical species A,B,D:
## subjected to equilibrium reaction D <-> A + B
## D is produced at a constant rate, prod
## B is consumed at 1st order rate, r
## Chemical problem formulation 1: ODE

## Dissociation constant
K <- 1

## parameters
pars <- c(ka = 1e6,     # forward rate
          r = 1,         # order rate
          prod = 0.1)

Fun_ODE <- function (t, y, pars)
{
  with (as.list(c(y, pars)), {
    ra <- ka*D            # forward rate
    rb <- ka/K*A*B        # backward rate

    ## rates of changes
    dD <- -ra + rb + prod
    dA <- ra - rb
    dB <- ra - rb - r*B
    return(list(dy = c(dA, dB, dD),
                CONC = A+B+D))
  })
}

## Chemical problem formulation 2: DAE
## 1. get rid of the fast reactions ra and rb by taking linear combinations:
## dD-dA = prod (res1) and
## dB-dA = -r*B (res2)
## 2. In addition, the equilibrium condition (eq) reads:
## as ra = rb : ka*D = ka/K*A*B = > K*D = A+B

Res_DAE <- function (t, y, yprime, pars)
{
  with (as.list(c(y, yprime, pars)), {
    # residuals of lumped rates of changes
    res1 <- -dD - dA + prod
    res2 <- -dB - dA + prod
    res_eq <- ra - rb - ka*D - ka/K*A*B
    return(list(res1 = res1, res2 = res2, res_eq = res_eq))
  })
}
```
res2 <- -dB + dA - r*B

## and the equilibrium equation
eq <- K*D - A*B

return(list(c(res1, res2, eq),
            CONC = A+B+D))
}

## Chemical problem formulation 3: Mass * Func
## Based on the DAE formulation

Mass_FUNC <- function(t, y, pars) {
  with(as.list(c(y, pars)), {

    ## as above, but without the
    f1 <- prod
    f2 <- - r*B

    ## and the equilibrium equation
    f3 <- K*D - A*B

    return(list(c(f1, f2, f3),
                CONC = A+B+D))
  })
}

Mass <- matrix(nrow = 3, ncol = 3, byrow = TRUE,
               data=c(1, 0, 1, # dA + 0 + dB
                      -1, 1, 0,  # -dA + dB +0
                      0, 0, 0)) # algebraic

times <- seq(0, 100, by = 2)

## Initial conc; D is in equilibrium with A,B
y <- c(A = 2, B = 3, D = 2*3/K)

## ODE model solved with daspk
ODE <- daspk(y = y, times = times, func = Fun_ODE,
              pars = pars, atol = 1e-10, rtol = 1e-10)

## Initial rate of change
dy <- c(dA = 0, dB = 0, dD = 0)

## DAE model solved with daspk
DAE <- daspk(y = y, dy = dy, times = times,
              res = Res_DAE, pars = pars, atol = 1e-10, rtol = 1e-10)

MASS <- daspk(y=y, times=times, func = Mass_FUNC, pars = pars, mass = Mass)

## ------------------------
## plotting output
### ---------------

```r
code
plot(ODE, DAE, xlab = "time", ylab = "conc", type = c("l", "p"),
     pch = c(NA, 1))

legend("bottomright", lty = c(1, NA), pch = c(NA, 1),
       col = c("black", "red"), legend = c("ODE", "DAE"))
```

# difference between both implementations:
```r
code
max(abs(ODE-DAE))
```

### ---

## same DAE model, now with the Jacobian
### --------------------------------------

```r
code
jacres_DAE <- function (t, y, yprime, pars, cj)
{
  with (as.list(c(y, yprime, pars)), {
    # res1 = -dD - dA + prod
    PD[1,1] <- -1*cj  # d(res1)/d(A)-cj*d(res1)/d(dA)
    PD[1,2] <- 0     # d(res1)/d(B)-cj*d(res1)/d(dB)
    PD[1,3] <- -1*cj # d(res1)/d(D)-cj*d(res1)/d(dD)
    # res2 = -dB + dA - r*B
    PD[2,1] <- 1*cj  # d(res2)/d(A)+cj*d(res2)/d(dA)
    PD[2,2] <- r    # d(res2)/d(B)+cj*d(res2)/d(dB)
    PD[2,3] <- 0     # d(res2)/d(D)+cj*d(res2)/d(dD)
    # eq = K*D - A*B
    PD[3,1] <- 0
    PD[3,2] <- -A
    PD[3,3] <- K
    return(PD)
  })
}
```

```r
code
PD <- matrix(ncol = 3, nrow = 3, 0)
```

```r
code
DAE2 <- daspk(y = y, dy = dy, times = times,
              res = Res_DAE, jacres = jacres_DAE, jactype = "fullusr",
              parms = pars, atol = 1e-10, rtol = 1e-10)
```

```r
code
max(abs(DAE-DAE2))
```

### See `\dynload subdirectory for a FORTRAN implementation of this model
### ------------------

## The chemical model as a DLL, with production a forcing function
### ---------------------------------------------------------------

```r
code
times <- seq(0, 100, by = 2)
pars <- c(K = 1, ka = 1e6, r = 1)
```

## Initial conc; D is in equilibrium with A,B
```r
code
y <- c(A = 2, B = 3, D = as.double(2*3/pars["K"]))
```
## Initial rate of change

```r
dy <- c(dA = 0, dB = 0, dD = 0)
```

## production increases with time

```r
prod <- matrix(ncol = 2,
              data = c(seq(0, 100, by = 10), 0.1*(1+runif(11)*1)))
```

```r
ODE_dll <- daspk(y = y, dy = dy, times = times, res = "chemres",
                 dllname = "deSolve", initfunc = "initparms",
                 initforc = "initforces", pars = pars, forcings = prod,
                 atol = 1e-10, rtol = 1e-10, nout = 2,
                 outnames = c("CONC","Prod"))
```

```r
plot(ODE_dll, which = c("Prod", "D"), xlab = "time",
     ylab = c("/day", "conc"), main = c("production rate","D"))
```

dede General Solver for Delay Differential Equations.

### Description

Function `dede` is a general solver for delay differential equations, i.e. equations where the derivative depends on past values of the state variables or their derivatives.

### Usage

```r
dede(y, times, func=NULL, pars,
      method = c("lsoda", "lsode", "lsodes", "lsodar", "vode",
                 "daspk", "bdf", "adams", "impAdams", "radau"), control = NULL, ...)
```

### Arguments

- **y**: the initial (state) values for the DE system, a vector. If `y` has a name attribute, the names will be used to label the output matrix.
- **times**: time sequence for which output is wanted; the first value of `times` must be the initial time.
- **func**: an R-function that computes the values of the derivatives in the ODE system (the model definition) at time `t`.
  - `func` must be defined as: `func <- function(t, y, pars, ...)`. `t` is the current time point in the integration, `y` is the current estimate of the variables in the DE system. If the initial values `y` has a names attribute, the names will be available inside `func`. `pars` is a vector or list of parameters; ... (optional) are any other arguments passed to the function.
  - The return value of `func` should be a list, whose first element is a vector containing the derivatives of `y` with respect to time, and whose next elements are
global values that are required at each point in times. The derivatives must be specified in the same order as the state variables y.

If method "daspk" is used, then func can be NULL, in which case res should be used.

parms parameters passed to func.

method the integrator to use, either a string ("lsoda", "lsode", "lsodes", "lsodar", "vode", "daspk", "bdf", "adams", "impAdams", "radau") or a function that performs the integration. The default integrator used is lsoda.

control a list that can supply (1) the size of the history array, as control$mxhist; the default is 1e4 and (2) how to interpolate, as control$interpol, where 1 is hermitian interpolation, 2 is variable order interpolation, using the Nordsieck history array. Only for the two Adams methods is the second option recommended.

... additional arguments passed to the integrator.

Details

Functions lagvalue and lagderiv are to be used with dede as they provide access to past (lagged) values of state variables and derivatives. The number of past values that are to be stored in a history matrix, can be specified in control$mxhist. The default value (if unspecified) is 1e4.

Cubic Hermite interpolation is used by default to obtain an accurate interpolant at the requested lagged time. For methods adams, impAdams, a more accurate interpolation method can be triggered by setting control$interpol = 2.

dede does not deal explicitly with propagated derivative discontinuities, but relies on the integrator to control the stepsize in the region of a discontinuity.

dede does not include methods to deal with delays that are smaller than the stepsize, although in some cases it may be possible to solve such models.

For these reasons, it can only solve rather simple delay differential equations.

When used together with integrator lsodar, or lsode, dde can simultaneously locate a root, and trigger an event. See last example.

Value

A matrix of class deSolve with up to as many rows as elements in times and as many columns as elements in y plus the number of "global" values returned in the second element of the return from func, plus an additional column (the first) for the time value. There will be one row for each element in times unless the integrator returns with an unrecoverable error. If y has a names attribute, it will be used to label the columns of the output value.

Author(s)

Karline Soetaert <karline.soetaert@nioz.nl>

See Also

lagvalue, lagderiv, for how to specify lagged variables and derivatives.
Examples

## A simple delay differential equation
## dy(t) = -y(t-1); y(t<0)=1

```r
#### the derivative function
####
derivs <- function(t, y, parms) {
  if (t <= 1)
    dy <- -1
  else
    dy <- - lagvalue(t - 1)
  list(c(dy))
}
```

#### initial values and times
####
yinit <- 1
times <- seq(0, 30, 0.1)

#### solve the model
####
yout <- dde(y = yinit, times = times, func = derivs, parms = NULL)

#### display, plot results
####
plot(yout, type = "l", lwd = 2, main = "dy/dt = -y(t-1)")

## The infectious disease model of Hairer; two lags.
## example 4 from Shampine and Thompson, 2000
## solving delay differential equations with dde23

```r
#### the derivative function
####
derivs <- function(t, y, parms) {
  if (t <= 1)
    lag1 <- 0.1
  else
    lag1 <- lagvalue(t - 1, 2)
  if (t <= 10)
    lag10 <- 0.1
  else
    lag10 <- lagvalue(t - 10, 2)
```
```r
dy1 <- -y[1] * lag1 + lag10
dy3 <- y[2] - lag10
list(c(dy1, dy2, dy3))
```

### initial values and times
```
yinit <- c(5, 0.1, 1)
times <- seq(0, 40, by = 0.1)
```

### solve the model
```
system.time(
  yout <- dede(y = yinit, times = times, func = derivs, parms = NULL)
)
```

### display, plot results
```
matplot(yout[,1], yout[,2], type = "l", lwd = 2, lty = 1,
  main = "Infectuous disease - Hairer")
```

### time lags + EVENTS triggered by a root function
```
The two-wheeled suitcase model
example 8 from Shampine and Thompson, 2000
solving delay differential equations with dde23
```

### the derivative function
```
derivs <- function(t, y, parms) {
  if (t < tau)
    lag <- 0
  else
    lag <- lagvalue(t - tau)

dy1 <- y[2]
dy2 <- -sign(y[1]) * gam * cos(y[1]) +
  sin(y[1]) - bet * lag[1] + A * sin(omega * t + mu)
  list(c(dy1, dy2))
}
```

### root and event function
```
root <- function(t,y,parms) ifelse(t>0, return(y), return(1))
event <- function(t,y,parms) return(c(y[1], y[2]+0.931))
```

```
gam = 0.248; bet = 1; tau = 0.1; A = 0.75
omega = 1.37; mu = asin(gam/A)
```
diagnostics

#-----------------------------
# initial values and times
#-----------------------------
yinit <- c(y = 0, dy = 0)
times <- seq(0, 12, len = 1000)

#-----------------------------
# solve the model
#-----------------------------
# Note: use a solver that supports both root finding and events,
# e.g. lsodar, lsode, lsoda, adams, bdf
yout <- dede(y = yinit, times = times, func = derivs, parms = NULL,
method = "lsodar", rootfun = root, events = list(func = event, root = TRUE))

#-----------------------------
# display, plot results
#-----------------------------
plot(yout, which = 1, type = "l", lwd = 2, main = "suitcase model", mfrow = c(1,2))
plot(yout[,2], yout[,3], xlab = "y", ylab = "dy", type = "l", lwd = 2)

---

diagnostics

Print Diagnostic Characteristics of Solvers

Description

Prints several diagnostics of the simulation to the screen, e.g. number of steps taken, the last step size, ...

Usage

diagnostics(obj, ...)

# Default S3 method:
diagnostics(obj, ...)

Arguments

obj is an output data structure produced by one of the solver routines.
...

optional arguments allowing to extend diagnostics as a generic function.

Details

Detailed information about the success of a simulation is printed, if a diagnostics function exists for a specific solver routine. A warning is printed, if no class-specific diagnostics exists.

Please consult the class-specific help page for details.
See Also
diagnostics.deSolve for diagnostics of differential equation solvers.

diagnostics.deSolve  Print Diagnostic Characteristics of ODE and DAE Solvers

Description
Prints several diagnostics of the simulation to the screen, e.g. number of steps taken, the last step size, ...

Usage
## S3 method for class 'deSolve'
diagnostics(obj, Full = FALSE, ...)

Arguments
- obj is the output matrix as produced by one of the integration routines.
- Full when TRUE then all messages will be printed, including the ones that are not relevant for the solver. If FALSE, then only the relevant messages will be printed.
- ... optional arguments allowing to extend diagnostics as a generic function.

Details
When the integration output is saved as a data.frame, then the required attributes are lost and method diagnostics will not work anymore.

Value
The integer and real vector with diagnostic values; for function lsodar also the root information. See tables 2 and 3 in vignette("deSolve") for what these vectors contain.

Note: the number of function evaluations are *without* the extra calls performed to generate the ordinary output variables (if present).

Examples
## The famous Lorenz equations: chaos in the earth's atmosphere

chaos <- function(t, state, parameters) {
  with(as.list(c(state)), {
    dx <- -8/3 * x + y * z
    dy <- -10 * (y - z)
    dz <- -x * y + 28 * y - z
  })
DLLfunc

Evaluates a Derivative Function Represented in a DLL

Description

Calls a function, defined in a compiled language as a DLL.

Usage

```r
DLLfunc(func, times, y, parms, dllname,
    initfunc = dllname, rpar = NULL, ipar = NULL, nout = 0,
    outnames = NULL, forcings = NULL, initforc = NULL,
    fcontrol = NULL)
```

Arguments

- `func`: the name of the function in the dynamically loaded shared library,
- `times`: first value = the time at which the function needs to be evaluated,
- `y`: the values of the dependent variables for which the function needs to be evaluated,
- `parms`: the parameters that are passed to the initialiser function,
- `dllname`: a string giving the name of the shared library (without extension) that contains the compiled function or subroutine definitions referred to in `func`,
- `initfunc`: if not NULL, the name of the initialisation function (which initialises values of parameters), as provided in ‘dllname’. See details.
- `rpar`: a vector with double precision values passed to the DLL-function `func` and `jacfunc` present in the DLL, via argument `rpar`,
- `ipar`: a vector with integer values passed to the dll-function `func` and `jacfunc` present in the DLL, via function argument `ipar`,
- `nout`: the number of output variables.
- `outnames`: only used if ‘dllname’ is specified and `nout > 0`: the names of output variables calculated in the compiled function `func`, present in the shared library.
forcings  only used if ‘dllname’ is specified: a list with the forcing function data sets, each present as a two-columned matrix, with (time, value); interpolation outside the interval \([\text{min(times)}, \text{max(times)}]\) is done by taking the value at the closest data extreme.

See package vignette "compiledCode".

initforc  if not NULL, the name of the forcing function initialisation function, as provided in ‘dllname’. It MUST be present if forcings has been given a value. See package vignette "compiledCode".

fcontrol  A list of control parameters for the forcing functions. See package vignette "compiledCode".

Details

This function is meant to help developing FORTRAN or C models that are to be used to solve ordinary differential equations (ODE) in packages deSolve and/or rootSolve.

Value

a list containing:

dy  the rate of change estimated by the function,

var  the ordinary output variables of the function.

Author(s)

Karline Soetaert <karline.soetaert@nioz.nl>

See Also

ode for a general interface to most of the ODE solvers

Examples

```
### ex. 1
### ccl4model
### Parameter values and initial conditions
### see example(ccl4model) for a more comprehensive implementation

Parms <- c(0.182, 4.0, 4.0, 0.08, 0.04, 0.74, 0.05, 0.15, 0.32,
           16.17, 281.48, 13.3, 16.17, 5.487, 153.8, 0.04321671,
           0.4027255, 1000, 0.02, 1.0, 3.8)

yini <- c(AI = 21, AAM = 0, AT = 0, AF = 0, AL = 0, CLT = 0, AM = 0)

### the rate of change
DLLfunc(y = yini, dllname = "deSolve", func = "derivccl4",
        initfunc = "initccl4", parms =Parms, times = 1,
        nout = 3, outnames = c("DOSE", "MASS", "CP") )
```
DLLres

Evaluates a Residual Derivative Function Represented in a DLL

Description

Calls a residual function, \( F(t, y, y') \) of a DAE system (differential algebraic equations) defined in a compiled language as a DLL.

To be used for testing the implementation of DAE problems in compiled code

Usage

\[
\text{DLLres}(\text{res, times, y, dy, parms, dllname,}}
\text{ initfunc = dllname, rpar = NULL, ipar = NULL, nout = 0,}
\text{ outnames = NULL, forcings = NULL, initforc = NULL,}
\text{ fcontrol = NULL})
\]

Arguments

- **res**: the name of the function in the dynamically loaded shared library.
- **times**: first value = the time at which the function needs to be evaluated,
- **y**: the values of the dependent variables for which the function needs to be evaluated,
- **dy**: the derivative of the values of the dependent variables for which the function needs to be evaluated,
parms      the parameters that are passed to the initialiser function,
dllname    a string giving the name of the shared library (without extension) that contains
the compiled function or subroutine definitions referred to in func,
initfunc   if not NULL, the name of the initialisation function (which initialises values of
parameters), as provided in ‘dllname’. See details,
rpar       a vector with double precision values passed to the DLL-function func and
jacfunc    present in the DLL, via argument rpar,
ipar       a vector with integer values passed to the DLL-function func and jacfunc
present in the DLL, via function argument ipar,
nout       the number of output variables.
outnames   only used if ‘dllname’ is specified and nout > 0: the names of output variables
calculated in the compiled function func, present in the shared library.
forcings  only used if ‘dllname’ is specified: a list with the forcing function data sets,
each present as a two-columned matrix, with (time,value); interpolation outside
the interval [min(times), max(times)] is done by taking the value at the closest
data extreme.
        See package vignette “compiledCode”.
initforc   if not NULL, the name of the forcing function initialisation function, as provided
in ‘dllname’. It MUST be present if forcings has been given a value. See
package vignette “compiledCode”.
fcontrol  A list of control parameters for the forcing functions. See package vignette
“compiledCode”.

Details
This function is meant to help developing FORTRAN or C models that are to be used to solve
differential algebraic equations (DAE) in package deSolve.

Value
a list containing:
res        the residual of derivative estimated by the function
var        the ordinary output variables of the function

Author(s)
Karline Soetaert <karline.soetaert@nioz.nl>

See Also
daspk to solve DAE problems
Implementing Events and Roots in Differential Equation Models.

Description

An event occurs when the value of a state variable is suddenly changed, e.g. because a value is added, subtracted, or multiplied. The integration routines cannot deal easily with such state variable changes. Typically these events occur only at specific times. In deSolve, events can be imposed by means of an input data.frame, that specifies at which time and how a certain state variable is altered, or via an event function.

Roots occur when a root function becomes zero. By default when a root is found, the simulation either stops (no event), or triggers an event.

Details

The events are specified by means of argument events passed to the integration routines. events should be a list that contains one of the following:

1. func: an R-function or the name of a function in compiled code that specifies the event,
2. data: a data.frame that specifies the state variables, times, values and types of the events. Note that the event times must also be part of the integration output times, else the event will not take place. As from version 1.9.1, this is checked by the solver, and a warning message is produced if event times are missing in times; see also cleanEventTimes for utility functions to check and solve such issues.
3. **time**: when events are specified by an event function: the times at which the events take place. Note that these event times must also be part of the integration output times exactly, else the event would not take place. As from version 1.9.1 this is checked by the solver, and an error message produced if event times are missing in times; see also `cleanEventTimes` for utility functions to check and solve such issues.

4. **root**: when events are specified by a function and triggered by a root, this logical should be set equal to `TRUE`.

5. **terminalroot** when events are triggered by a root, the default is that the simulation continues after the event is executed. In `terminalroot`, we can specify which roots should terminate the simulation.

6. **maxroot** when `root = TRUE`, the maximal number of times at with a root is found and that are kept; defaults to 100. If the number of roots > `maxroot`, then only the first `maxroot` will be outputted.

7. **ties**: if events, as specified by a data.frame are "ordered", set to "ordered", the default is "notordered". This will save some computational time.

In case the events are specified by means of an **R function** (argument `events$func`), it must be defined as: `function(t, y, parms, ...)`. `t` is the current time point in the integration, `y` is the current estimate of the variables in the ODE system. If the initial values `y` has a names attribute, the names will be available inside `events$func`. `parms` is a vector or list of parameters; `...` (optional) are any other arguments passed to the function via the call to the integration method. The event function should return the `y`-values (some of which modified), as a **vector**.

If `events$func` is a string, this indicates that the events are specified by a function in compiled code. This function has as arguments, the number of state variables, the time, and the state variable vector. See package vignette "compiledCode" for more details.

In case events are specified by an **R-function**, this requires either: input of the **time** of the events, a vector as defined in `events$time` **OR** the specification of a **root** function. In the latter case, the model must be solved with an integration routine with root-finding capability.

The root function itself should be specified with argument `rootfunc`. In this case, the integrator is informed that the simulation to be continued after a root is found by setting `events$root` equal to `TRUE`.

If the events are specified by a **data frame** (argument `events$data`), this should contain the following columns (and in that order):

1. **var** the state variable name or number that is affected by the event
2. **time** the time at which the event is to take place; the solvers will check if the time is embraced by the simulation time
3. **value** the value, magnitude of the event
4. **method** which event is to take place; should be one of ("replace", "add", "multiply"); also allowed is to specify the number (1 = replace, 2 = add, 3 = multiply)

For instance, the following line

```
v1 10 2 "add"
```

will cause the value 2 to be added to a state variable, called "v1" at time = 10.

From deSolve version 1.9.1 the following routines have **root-finding** capability: `lsoda`, `lsode`, `lsodes`, and `radau`. For the first 3 integration methods, the root finding algorithm is based on the
events

algorithm in solver LSODAR, and is implemented in FORTRAN. For radau, the root solving algorithm is written in C-code, and it works slightly different. Thus, some problems involving roots may be more efficiently solved with either lsoda, lsode, or lsodes, while other problems are more efficiently solved with radau.

If a root function is defined, but not an event function, then by default the solver will stop at a root. If this is not desirable, e.g. because we want to record the position of many roots, then a dummy "event" function can be defined which returns the values of the state variables - unaltered.

If roots and events are combined, and roots are found, then the output will have attribute troot which will contain the times at which a root was found (and the event triggered). There will be at most eventsDmaxroot such values. The default is 100.

See two last examples; also see example of ccl4model.

Author(s)

Karline Soetaert

See Also

forcings, for how to implement forcing functions.
lsodar, for more examples of roots

Examples

```r
## 1. EVENTS in a data.frame

# derivative function: derivatives set to 0
derivs <- function(t, var, parms) {
    list(dvar = rep(0, 2))
}

yini <- c(v1 = 1, v2 = 2)
times <- seq(0, 10, by = 0.1)

eventdat <- data.frame(var = c("v1", "v2", "v2", "v1"),
                        time = c(1, 1, 5, 9),
                        value = c(1, 2, 3, 4),
                        method = c("add", "mult", "rep", "add"))

out <- vode(func = derivs, y = yini, times = times, parms = NULL,
            events = list(data = eventdat))
plot(out)

# eventdat <- data.frame(var = c(rep("v1", 10), rep("v2", 10)),
#                         time = c(1:10, 1:10),
#                         value = runif(20),
#                         method = rep("add", 20))
```
eventdat

out <- ode(func = derivs, y = yini, times = times, parms = NULL,
            events = list(data = eventdat))

plot(out)

### 2. EVENTS in a function
### 
### derivative function: rate of change v1 = 0, v2 reduced at first-order rate
derivs <- function(t, var, parms) {
  list(c(0, -0.5 * var[2]))
}

# events: add 1 to v1, multiply v2 with random number
eventfun <- function(t, y, parms){
  with(as.list(y),{
    v1 <- v1 + 1
    v2 <- 5 * runif(1)
    return(c(v1, v2))
  })
}

yini <- c(v1 = 1, v2 = 2)
times <- seq(0, 10, by = 0.1)

out <- ode(func = derivs, y = yini, times = times, parms = NULL,
            events = list(func = eventfun, time = c(1:9, 2.2, 2.4)))
plot(out, type = "l")

### 3. EVENTS triggered by a root function
### 
### derivative: simple first-order decay
derivs <- function(t, y, pars) {
  return(list(-0.1 * y))
}

# event triggered if state variable = 0.5
rootfun <- function(t, y, pars) {
  return(y - 0.5)
}

# sets state variable = 1
eventfun <- function(t, y, pars) {
  return(y = 1)
}

yini <- 2
events

```r

# times <- seq(0, 100, 0.1)

## uses ode to solve; root = TRUE specifies that the event is
## triggered by a root.
out <- ode(times = times, y = yini, func = derivs, parms = NULL,
   events = list(func = eventfun, root = TRUE),
   rootfun = rootfun)

plot(out, type = "l")

## time of the root:
troot <- attributes(out)$troot
points(troot, rep(0.5, length(troot)))

## 4. More ROOT examples: Rotation function

## Rotate <- function(t, x, p)
##   list(c( x[2],
##          -x[1] ))

## Root = when second state variable = 0
rootfun <- function(t, x, p) x[2]

## "event" returns state variables unchanged
eventfun <- function(t, x, p) x
times <- seq(from = 0, to = 15, by = 0.1)

## 1. No event: stops at first root
out1 <- ode(func = Rotate, y = c(5, 5), parms = 0,
   times = times, rootfun = rootfun)
tail(out1)

## 2. Continues till end of times and records the roots
out <- ode(func = Rotate, y = c(5, 5), parms = 0,
   times = times, rootfun = rootfun,
   events = list(func = eventfun, root = TRUE) )

plot(out)
troot <- attributes(out)$troot # time of roots
points(troot, rep(0, length(troot)))

## Multiple roots: either one of the state variables = 0
root2 <- function(t, x, p) x

out2 <- ode(func = Rotate, y = c(5, 5), parms = 0,
   times = times, rootfun = root2,
   events = list(func = eventfun, root = TRUE) )

plot(out2, which = 2)
troot <- attributes(out2)$troot
indroot <- attributes(out2)$indroot # which root was found
```
```r
points(troot, rep(0, length(troot)), col = indroot, pch = 18, cex = 2)

## Multiple roots and stop at first time root 1.
out3 <- ode(func = Rotate, y = c(5, 5), parms = 0,
    times = times, rootfun = root2,
    events = list(func = eventfun, root = TRUE, terminalroot = 1))

## 5. Stop at 5th root - only works with radau.
## 
Rotate <- function(t, x, p)
    list(c( x[2],
          -x[1],
          0 ))

## Root = when second state variable = 0
root3 <- function(t, x, p) c(x[2], x[3] - 5)
event3 <- function(t, x, p) c(x[1:2], x[3]+1)
times <- seq(0, 15, 0.1)
out3 <- ode(func = Rotate, y = c(x1 = 5, x2 = 5, nroot = 0),
    parms = 0, method = "radau",
    times = times, rootfun = root3,
    events = list(func = event3, root = TRUE, terminalroot = 2))
plot(out3)
attributes(out3)[c("troot", "nroot", "indroot")]

## 6 Event in R-code, model function in compiled code - based on vode example
## 
times <- 1:365
Flux <- cbind(times, sin(pi*times/365)^2) # forcing function

# run without events
out <- ode(y = c(C = 1), times, func = "scocder", parms = c(k=0.01),
    dllname = "desolve", initforc = "scocforc", forcings = Flux,
    initfunc = "scocpar", nout = 2, outnames = c("Mineralisation", "Depo"))

# Event halves the concentration
EventMin <- function(t, y, p) y/2

out2 <- ode(y = c(C = 1), times, func = "scocder", parms = c(k=0.01),
    dllname = "desolve", initforc = "scocforc", forcings = Flux,
    initfunc = "scocpar", nout = 2, outnames = c("Mineralisation", "Depo"),
    events = list(func = EventMin, time = c(50.1, 200, 210.5)))

plot(out, out2)
```
Description
A forcing function is an external variable that is essential to the model, but not explicitly modeled. Rather, it is imposed as a time-series. Thus, if a model uses forcing variables, their value at each time point needs to be estimated by interpolation of a data series.

Details
The forcing functions are imposed as a data series, that contains the values of the forcings at specified times.

Models may be defined in compiled C or FORTRAN code, as well as in R.

If the model is defined in R code, it is most efficient to:
1. define a function that performs the linear interpolation, using R’s `approxfun`. It is generally recommended to use `rule = 2`, such as to allow extrapolation outside of the time interval, especially when using the Livermore solvers, as these may exceed the last time point.
2. call this function within the model’s derivative function, to interpolate at the current timestep.

See first example.

If the models are defined in compiled C or FORTRAN code, it is possible to use `desolve`’s forcing function update algorithm. This is the compiled-code equivalent of `approxfun` or `approx`.

In this case:
1. the forcing function data series is provided by means of argument `forcings`,
2. `initforc` is the name of the forcing function initialisation function, as provided in ‘dllname’, while
3. `fcontrol` is a list used to finetune how the forcing update should be performed.

The `fcontrol` argument is a list that can supply any of the following components (conform the definitions in the `approxfun` function):

- **method** specifies the interpolation method to be used. Choices are "linear" or "constant",
- **rule** an integer describing how interpolation is to take place outside the interval [min(times), max(times)]. If `rule` is 1 then an error will be triggered and the calculation will stop if `times` extends the interval of the forcing function data set. If it is 2, the **default**, the value at the closest data extreme is used, a warning will be printed if `verbose` is `TRUE`.
  Note that the default differs from the `approx` default.
- **f** For `method = "constant"` a number between 0 and 1 inclusive, indicating a compromise between left- and right-continuous step functions. If `y0` and `y1` are the values to the left and right of the point then the value is `y0 * (1 - f) + y1 * f` so that `f = 0` is right-continuous and `f = 1` is left-continuous,
ties Handling of tied times values. Either a function with a single vector argument returning a single number result or the string "ordered".

Note that the default is "ordered", hence the existence of ties will NOT be investigated; in the C code this will mean that -if ties exist, the first value will be used; if the dataset is not ordered, then nonsense will be produced.

Alternative values for ties are mean, min etc

The defaults are:

```r
fcontrol = list(method = "linear", rule = 2, f = 0, ties = "ordered")
```

Note that only ONE specification is allowed, even if there is more than one forcing function data set.

More information about models defined in compiled code is in the package vignette ("compiled-Code").

Note

How to write compiled code is described in package vignette "compiledCode", which should be referred to for details.

This vignette also contains examples on how to pass forcing functions.

Author(s)

Karline Soetaert,
Thomas Petzoldt,
R. Woodrow Setzer

See Also

approx or approxfun, the R function,

events for how to implement events.

Examples

```r
## FORCING FUNCTION: The sediment oxygen consumption example - R-code:

## Forcing function data
Flux <- matrix(ncol=2,byrow=TRUE,data=c(
  1, 0.654, 11, 0.167, 21, 0.060, 41, 0.070, 73, 0.277, 83, 0.186,
  93, 0.140, 103, 0.255, 113, 0.231, 123, 0.309, 133, 1.127, 143, 1.923,
  153, 1.091, 163, 1.001, 173, 1.691, 163, 1.404, 194, 1.226, 204, 0.767,
  214, 0.893, 224, 0.737, 234, 0.772, 244, 0.726, 254, 0.624, 264, 0.439,
  274, 0.168, 284, 0.280, 294, 0.202, 304, 0.193, 315, 0.286, 325, 0.599,
  335, 1.889, 345, 0.996, 355, 0.681, 365, 1.135))

parms <- c(k=0.01)
```
times <- 1:365

## the model
sediment <- function(t, O2, k)
  list(c(Depo(t) - k * O2), depo = Depo(t))

# the forcing functions; rule = 2 avoids NaNs in interpolation
Depo <- approxfun(x = Flux[,1], y = Flux[,2], method = "linear", rule = 2)

Out <- ode(times = times, func = sediment, y = c(O2 = 63), parms = parms)

## same forcing functions, now constant interpolation
Depo <- approxfun(x = Flux[,1], y = Flux[,2], method = "constant",
  f = 0.5, rule = 2)

Out2 <- ode(times = times, func = sediment, y = c(O2 = 63), parms = parms)

mf <- par(mfrow = c(2, 1))
plot(Out, which = "depo", type = "l", lwd = 2, mfrow = NULL)
lines(Out2[,"time"], Out2[,"depo"], col = "red", lwd = 2)

plot(Out, which = "O2", type = "l", lwd = 2, mfrow = NULL)
lines(Out2[,"time"], Out2[,"O2"], col = "red", lwd = 2)

## SCOC is the same model, as implemented in FORTRAN

out<- SCOC(times, parms = parms, Flux = Flux)

plot(out[,"time"], out[,"Depo"], type = "l", col = "red")
lines(out[,"time"], out[,"Mineralisation"], col = "blue")

## Constant interpolation of forcing function - left side of interval
fcontrol <- list(method = "constant")

out2 <- SCOC(times, parms = parms, Flux = Flux, fcontrol = fcontrol)

plot(out2[,"time"], out2[,"Depo"], type = "l", col = "red")
lines(out2[,"time"], out2[,"Mineralisation"], col = "blue")

## Not run:
## show examples (see respective help pages for details)

example(aquaphy)

## show package vignette with tutorial about how to use compiled models
## + source code of the vignette
## + directory with C and FORTRAN sources
vignette("compiledCode")
Description

Solving initial value problems for stiff or non-stiff systems of first-order ordinary differential equations (ODEs).

The R function `lsoda` provides an interface to the FORTRAN ODE solver of the same name, written by Linda R. Petzold and Alan C. Hindmarsh.

The system of ODE’s is written as an R function (which may, of course, use `.C`, `.Fortran`, `.Call`, etc., to call foreign code) or be defined in compiled code that has been dynamically loaded. A vector of parameters is passed to the ODEs, so the solver may be used as part of a modeling package for ODEs, or for parameter estimation using any appropriate modeling tool for non-linear models in R such as `optim`, `nls`, `nlm` or `nlme`.

`lsoda` differs from the other integrators (except `lsodar`) in that it switches automatically between stiff and nonstiff methods. This means that the user does not have to determine whether the problem is stiff or not, and the solver will automatically choose the appropriate method. It always starts with the nonstiff method.

Usage

```r
lsoda(y, times, func, parms, rtol = 1e-6, atol = 1e-6, 
  jacfunc = NULL, jactype = "fullint", rootfunc = NULL, 
  verbose = FALSE, nroot = 0, tcrit = NULL, 
  hmin = 0, hmax = NULL, hini = 0, ynames = TRUE, 
  maxordn = 12, maxords = 5, bandup = NULL, banddown = NULL, 
  maxsteps = 5000, dllname = NULL, initfunc = dllname, 
  initpar = parms, rpar = NULL, ipar = NULL, nout = 0, 
  outnames = NULL, forcings = NULL, initforc = NULL, 
  fcontrol = NULL, events = NULL, lags = NULL,...)
```

Arguments

- `y`: the initial (state) values for the ODE system. If `y` has a name attribute, the names will be used to label the output matrix.
- `times`: times at which explicit estimates for `y` are desired. The first value in `times` must be the initial time.
func either an \texttt{R}-function that computes the values of the derivatives in the ODE system (the \textit{model definition}) at time \( t \), or a character string giving the name of a compiled function in a dynamically loaded shared library.

If \texttt{func} is an \texttt{R}-function, it must be defined as: \texttt{func \leftarrow function(t, y, parms, ...).} \( t \) is the current time point in the integration, \( y \) is the current estimate of the variables in the ODE system. If the initial values \( y \) has a names attribute, the names will be available inside \texttt{func}. \texttt{parms} is a vector or list of parameters; ... (optional) are any other arguments passed to the function.

The return value of \texttt{func} should be a list, whose first element is a vector containing the derivatives of \( y \) with respect to time, and whose next elements are global values that are required at each point in \texttt{times}. The derivatives must be specified in the \textbf{same order} as the state variables \( y \).

If \texttt{func} is a string, then \texttt{dllname} must give the name of the shared library (without extension) which must be loaded before \texttt{lsoda()} is called. See package vignette "\texttt{compiledCode}" for more details.

\begin{itemize}
  \item \texttt{parms} vector or list of parameters used in \texttt{func} or \texttt{jacfunc}.
  \item \texttt{rtol} relative error tolerance, either a scalar or an array as long as \( y \). See details.
  \item \texttt{atol} absolute error tolerance, either a scalar or an array as long as \( y \). See details.
  \item \texttt{jacfunc} if not \texttt{NULL}, an \texttt{R} function, that computes the Jacobian of the system of differential equations \( \partial y_i / \partial y_j \), or a string giving the name of a function or subroutine in \texttt{\'dllname\'} that computes the Jacobian (see vignette "\texttt{compiledCode}" for more about this option).
    In some circumstances, supplying \texttt{jacfunc} can speed up the computations, if the system is stiff. The \texttt{R} calling sequence for \texttt{jacfunc} is identical to that of \texttt{func}.
    If the Jacobian is a full matrix, \texttt{jacfunc} should return a matrix \( \partial y_i / \partial y_j \), where the \( i \)th row contains the derivative of \( dy_i / dt \) with respect to \( y_j \), or a vector containing the matrix elements by columns (the way \texttt{R} and FORTRAN store matrices).
    If the Jacobian is banded, \texttt{jacfunc} should return a matrix containing only the nonzero bands of the Jacobian, rotated row-wise. See first example of \texttt{lsode}.
    \texttt{jactype} the structure of the Jacobian, one of "\texttt{fullint}", "\texttt{fullusr}", "\texttt{bandusr}" or "\texttt{bandint}" - either full or banded and estimated internally or by user.
  \item \texttt{rootfunc} if not \texttt{NULL}, an \texttt{R} function that computes the function whose root has to be estimated or a string giving the name of a function or subroutine in \texttt{\'dllname\'} that computes the root function. The \texttt{R} calling sequence for \texttt{rootfunc} is identical to that of \texttt{func}. \texttt{rootfunc} should return a vector with the function values whose root is sought. When \texttt{rootfunc} is provided, then \texttt{lsodar} will be called.
  \item \texttt{verbose} if \texttt{TRUE}: full output to the screen, e.g. will print the diagnostics of the integration - see details.
  \item \texttt{nroot} only used if \texttt{\'dllname\'} is specified: the number of constraint functions whose roots are desired during the integration; if \texttt{rootfunc} is an \texttt{R}-function, the solver estimates the number of roots.
  \item \texttt{tcrit} if not \texttt{NULL}, then \texttt{lsoda} cannot integrate past \texttt{tcrit}. The FORTRAN routine \texttt{lsoda} overshoots its targets (times points in the vector \texttt{times}), and interpolates values for the desired time points. If there is a time beyond which integration
\end{itemize}
should not proceed (perhaps because of a singularity), that should be provided in `tcrit`.

- **hmin**: an optional minimum value of the integration stepsize. In special situations this parameter may speed up computations with the cost of precision. Don’t use `hmin` if you don’t know why!

- **hmax**: an optional maximum value of the integration stepsize. If not specified, `hmax` is set to the largest difference in `times`, to avoid that the simulation possibly ignores short-term events. If 0, no maximal size is specified.

- **hini**: initial step size to be attempted; if 0, the initial step size is determined by the solver.

- **ynames**: logical, if `FALSE`: names of state variables are not passed to function `func`; this may speed up the simulation especially for large models.

- **maxordn**: the maximum order to be allowed in case the method is non-stiff. Should be <= 12. Reduce `maxord` to save storage space.

- **maxords**: the maximum order to be allowed in case the method is stiff. Should be <= 5. Reduce maxord to save storage space.

- **bandup**: number of non-zero bands above the diagonal, in case the Jacobian is banded.

- **banddown**: number of non-zero bands below the diagonal, in case the Jacobian is banded.

- **maxsteps**: maximal number of steps per output interval taken by the solver.

- **dllname**: a string giving the name of the shared library (without extension) that contains all the compiled function or subroutine definitions referred to in `func` and `jacfunc`. See package vignette "compiledCode".

- **initfunc**: if not `NULL`, the name of the initialisation function (which initialises values of parameters), as provided in `dllname`. See package vignette "compiledCode".

- **initpar**: only when `dllname` is specified and an initialisation function `initfunc` is in the dll: the parameters passed to the initialiser, to initialise the common blocks (FORTRAN) or global variables (C, C++).

- **rpar**: only when `dllname` is specified: a vector with double precision values passed to the dll-functions whose names are specified by `func` and `jacfunc`.

- **ipar**: only when `dllname` is specified: a vector with integer values passed to the dll-functions whose names are specified by `func` and `jacfunc`.

- **nout**: only used if `dllname` is specified and the model is defined in compiled code: the number of output variables calculated in the compiled function `func`, present in the shared library. Note: it is not automatically checked whether this is indeed the number of output variables calculated in the dll - you have to perform this check in the code. See package vignette "compiledCode".

- **outnames**: only used if `dllname` is specified and `nout > 0`: the names of output variables calculated in the compiled function `func`, present in the shared library. These names will be used to label the output matrix.

- **forcings**: only used if `dllname` is specified: a list with the forcing function data sets, each present as a two-columned matrix, with (time,value); interpolation outside the interval [min(`times`), max(`times`)] is done by taking the value at the closest data extreme. See `forcings` or package vignette "compiledCode".
if not NULL, the name of the forcing function initialisation function, as provided in 'dllname'. It MUST be present if forcings has been given a value. See forcings or package vignette "compiledCode".

fcontrol A list of control parameters for the forcing functions. See forcings or vignette compiledCode.

events A matrix or data frame that specifies events, i.e. when the value of a state variable is suddenly changed. See events for more information.

lags A list that specifies timelags, i.e. the number of steps that has to be kept. To be used for delay differential equations. See timelags, dede for more information.

Details

All the hard work is done by the FORTRAN subroutine lsoda, whose documentation should be consulted for details (it is included as comments in the source file 'src/opkdmain.f'). The implementation is based on the 12 November 2003 version of lsoda, from Netlib.

lsoda switches automatically between stiff and nonstiff methods. This means that the user does not have to determine whether the problem is stiff or not, and the solver will automatically choose the appropriate method. It always starts with the nonstiff method.

The form of the Jacobian can be specified by jactype which can take the following values:

"fullint" a full Jacobian, calculated internally by lsoda, the default,
"fullusr" a full Jacobian, specified by user function jacfunc,
"bandusr" a banded Jacobian, specified by user function jacfunc the size of the bands specified by bandup and banddown,
"bandint" banded Jacobian, calculated by lsoda; the size of the bands specified by bandup and banddown.

If jactype = "fullusr" or "bandusr" then the user must supply a subroutine jacfunc.

The following description of error control is adapted from the documentation of the lsoda source code (input arguments rtol and atol, above):

The input parameters rtol, and atol determine the error control performed by the solver. The solver will control the vector e of estimated local errors in y, according to an inequality of the form max-norm of (e/ewt) ≤ 1, where ewt is a vector of positive error weights. The values of rtol and atol should all be non-negative. The form of ewt is:

$$ rtol \times \text{abs}(y) + atol $$

where multiplication of two vectors is element-by-element.

If the request for precision exceeds the capabilities of the machine, the FORTRAN subroutine lsoda will return an error code; under some circumstances, the R function lsoda will attempt a reasonable reduction of precision in order to get an answer. It will write a warning if it does so.

The diagnostics of the integration can be printed to screen by calling diagnostics. If verbose = TRUE, the diagnostics will written to the screen at the end of the integration.
See vignette("deSolve") for an explanation of each element in the vectors containing the diagnostic properties and how to directly access them.

Models may be defined in compiled C or FORTRAN code, as well as in an R-function. See package vignette "compiledCode" for details.

More information about models defined in compiled code is in the package vignette ("compiled-Code"); information about linking forcing functions to compiled code is in forcings.

Examples in both C and FORTRAN are in the ‘dynload’ subdirectory of the deSolve package directory.

**Value**

A matrix of class deSolve with up to as many rows as elements in `times` and as many columns as elements in `y` plus the number of "global" values returned in the next elements of the return from `func`, plus and additional column for the time value. There will be a row for each element in `times` unless the FORTRAN routine ‘lsoda’ returns with an unrecoverable error. If `y` has a names attribute, it will be used to label the columns of the output value.

**Note**

The ‘demo’ directory contains some examples of using gnls to estimate parameters in a dynamic model.

**Author(s)**

R. Woodrow Setzer <setzer.woodrow@epa.gov>

**References**


Netlib: [http://www.netlib.org](http://www.netlib.org)

**See Also**

- `rk`, `rkMethod`, `rk4` and `euler` for Runge-Kutta integrators.
- `lsode`, which can also find a root
- `lsodes`, `lsodar`, `vode`, `daspk` for other solvers of the Livermore family,
- `ode` for a general interface to most of the ODE solvers,
- `ode.band` for solving models with a banded Jacobian,
- `ode.1D` for integrating 1-D models,
- `ode.2D` for integrating 2-D models,
- `ode.3D` for integrating 3-D models,
- `diagnostics` to print diagnostic messages.
Examples

```r
SPCmod <- function(t, x, parms) {
  with(as.list(c(parms, x)), {
    import <- sigimp(t)
    dS <- import - b*S*P + g*C  # substrate
    dP <- c*S*P - d*C*P        # producer
    dC <- e*C*P - f*C          # consumer
    res <- c(dS, dP, dC)
    list(res)
  })
}
```

## Parameters
parms <- c(b = 0.0, c = 0.1, d = 0.1, e = 0.1, f = 0.1, g = 0.0)

## vector of timesteps
times <- seq(0, 100, length = 101)

## external signal with rectangle impulse
signal <- as.data.frame(list(times = times,
                              import = rep(0, length(times))))

signal$import[signal$times >= 10 & signal$times <= 11] <- 0.2

sigimp <- approxfun(signal$times, signal$import, rule = 2)

## Start values for steady state
y <- xstart <- c(S = 1, P = 1, C = 1)

## Solving
out <- lsoda(xstart, times, SPCmod, parms)

## Plotting
mf <- par("mfrow")
plot(out, main = c("substrate", "producer", "consumer"))
plot(out[, "P"], out[, "C"], type = "l", xlab = "producer", ylab = "consumer")
par(mfrow = mf)
```

## Notes:
1. parameter and state variable names made accessible via "with" function
2. function sigimp accessible through lexical scoping
(see also ode and rk examples)
## from lsoda source code

### names makes this easier to read, but may slow down execution.

```r
c0 <- c(k1 = 0.04, k2 = 1e4, k3 = 3e7)
c1 <- c(1e-6, 1e-10, 1e-6)
times <- c(0, 4 * 10^(-(1:10)))
```

```r
list(c(d1, -d1-d3, d3), c(massbalance = sum(y)))
```

```r
d3 <- function(t, y, p) {
  matrix(c(-p[k1], p[k1], 0,
           2 * p[k3] * y[2],
           p[k2] * y[2], -p[k2] * y[2], 0)
         , 3, 3)
}
```

```
## measure speed (here and below)

```r
system.time(
  out <- lsoda(c(0, 0, 0), times, lsexamp, parms, rtol = 1e-4,
             atol = my.atol, hmax = Inf)
)
```

```r
out
```

### This is what the authors of lsoda got for the example:

#### the output of this program (on a cdc-7600 in single precision)

#### is as follows...

```r
## Using the analytic Jacobian speeds up execution a little:
```
lsodar

Solver for Ordinary Differential Equations (ODE), Switching Automatically Between Stiff and Non-stiff Methods and With Root Finding

Description

Solving initial value problems for stiff or non-stiff systems of first-order ordinary differential equations (ODEs) and including root-finding.

The R function lsodar provides an interface to the FORTRAN ODE solver of the same name, written by Alan C. Hindmarsh and Linda R. Petzold.

The system of ODE’s is written as an R function or be defined in compiled code that has been dynamically loaded. - see description of lsoda for details.

lsodar differs from lsode in two respects.

- It switches automatically between stiff and nonstiff methods (similar as lsoda).
- It finds the root of at least one of a set of constraint functions g(i) of the independent and dependent variables.

Two uses of lsodar are:

- To stop the simulation when a certain condition is met
- To trigger events, i.e. sudden changes in one of the state variables when a certain condition is met.

when a particular condition is met.

Usage

lsodar(y, times, func, parms, rtol = 1e-6, atol = 1e-6,
       jacfunc = NULL, jactype = "fullint", rootfunc = NULL,
       verbose = FALSE, nroot = 0, tcrit = NULL, hmin = 0,
       hmax = NULL, hini = 0, ynames = TRUE, maxordn = 12,
       maxords = 5, bandup = NULL, banddown = NULL, maxsteps = 5000,
       dllname = NULL, initfunc = dllname, initpar = parms,
       rpar = NULL, ipar = NULL, nout = 0, outnames = NULL, forcings=NULL,
       initforc = NULL, fcontrol=NULL, events=NULL, lags = NULL, ...)

system.time(
  outJ <- lsoda(c(1, 0, 0), times, lsexamp, parms, rtol = 1e-4,
                 atol = my.atol, jacfunc = exampjac, jactype = "fullusr", hmax = Inf)
)

all.equal(as.data.frame(out), as.data.frame(outJ)) # TRUE
diagnostics(out)
diagnostics(outJ) # shows what lsoda did internally
Arguments

**y**
the initial (state) values for the ODE system. If y has a name attribute, the names will be used to label the output matrix.

times
times at which explicit estimates for y are desired. The first value in times must be the initial time.

**func**
either an R-function that computes the values of the derivatives in the ODE system (the model definition) at time t, or a character string giving the name of a compiled function in a dynamically loaded shared library.

If func is an R-function, it must be defined as: `func <- function(t, y, parms, ...)`. t is the current time point in the integration, y is the current estimate of the variables in the ODE system. If the initial values y has a names attribute, the names will be available inside func. parms is a vector or list of parameters; ... (optional) are any other arguments passed to the function.

The return value of func should be a list, whose first element is a vector containing the derivatives of y with respect to time, and whose next elements are global values that are required at each point in times. The derivatives must be specified in the same order as the state variables y.

If func is a string, then dllname must give the name of the shared library (without extension) which must be loaded before lsodar() is called. See package vignette "compiledCode" for more details.

**parms**
vector or list of parameters used in func or jacfunc.

**rtol**
relative error tolerance, either a scalar or an array as long as y. See details.

**atol**
absolute error tolerance, either a scalar or an array as long as y. See details.

**jacfunc**
if not NULL, an R function, that computes the Jacobian of the system of differential equations $\partial \dot{y}_i / \partial y_j$, or a string giving the name of a function or subroutine in 'dllname' that computes the Jacobian (see vignette "compiledCode" for more about this option).

In some circumstances, supplying jacfunc can speed up the computations, if the system is stiff. The R calling sequence for jacfunc is identical to that of func.

If the Jacobian is a full matrix, jacfunc should return a matrix $\partial \dot{y}_i / \partial y_j$, where the ith row contains the derivative of $\dot{y}_i$ with respect to $y_j$, or a vector containing the matrix elements by columns (the way R and FORTRAN store matrices).

If the Jacobian is banded, jacfunc should return a matrix containing only the nonzero bands of the Jacobian, rotated row-wise. See first example of lsode.

**jactype**
the structure of the Jacobian, one of "fullint", "fullusr", "bandusr" or "bandint" - either full or banded and estimated internally or by user.

**rootfunc**
if not NULL, an R function that computes the function whose root has to be estimated or a string giving the name of a function or subroutine in 'dllname' that computes the root function. The R calling sequence for rootfunc is identical to that of func. rootfunc should return a vector with the function values whose root is sought.

**verbose**
a logical value that, when TRUE, will print the diagnostics of the integration - see details.
**nroot**

only used if ‘dllname’ is specified: the number of constraint functions whose roots are desired during the integration; if rootfunc is an R-function, the solver estimates the number of roots.

**tcrit**

if not NULL, then lsodar cannot integrate past tcrit. The FORTRAN routine lsodar overshoots its targets (times points in the vector times), and interpolates values for the desired time points. If there is a time beyond which integration should not proceed (perhaps because of a singularity), that should be provided in tcrit.

**hmin**

an optional minimum value of the integration stepsize. In special situations this parameter may speed up computations with the cost of precision. Don’t use hmin if you don’t know why!

**hmax**

an optional maximum value of the integration stepsize. If not specified, hmax is set to the largest difference in times, to avoid that the simulation possibly ignores short-term events. If 0, no maximal size is specified.

**hini**

initial step size to be attempted; if 0, the initial step size is determined by the solver.

**ynames**

logical, if FALSE: names of state variables are not passed to function func; this may speed up the simulation especially for large models.

**maxordn**

the maximum order to be allowed in case the method is non-stiff. Should be <= 12. Reduce maxord to save storage space.

**maxords**

the maximum order to be allowed in case the method is stiff. Should be <= 5. Reduce maxord to save storage space.

**bandup**

number of non-zero bands above the diagonal, in case the Jacobian is banded.

**banddown**

number of non-zero bands below the diagonal, in case the Jacobian is banded.

**maxsteps**

maximal number of steps per output interval taken by the solver.

**dllname**

a string giving the name of the shared library (without extension) that contains all the compiled function or subroutine definitions refered to in func and jacfunc. See package vignette “compiledCode”.

**initfunc**

if not NULL, the name of the initialisation function (which initialises values of parameters), as provided in ‘dllname’. See package vignette “compiledCode”.

**initpar**

only when ‘dllname’ is specified and an initialisation function initfunc is in the dll: the parameters passed to the initialiser, to initialise the common blocks (FORTRAN) or global variables (C, C++).

**rpar**

only when ‘dllname’ is specified: a vector with double precision values passed to the dll-functions whose names are specified by func and jacfunc.

**ipar**

only when ‘dllname’ is specified: a vector with integer values passed to the dll-functions whose names are specified by func and jacfunc.

**nout**

only used if dllname is specified and the model is defined in compiled code: the number of output variables calculated in the compiled function func, present in the shared library. Note: it is not automatically checked whether this is indeed the number of output variables calculated in the dll - you have to perform this check in the code - See package vignette “compiledCode”.
outnames only used if ‘dllname’ is specified and nout > 0: the names of output variables calculated in the compiled function func, present in the shared library. These names will be used to label the output matrix.

forcings only used if ‘dllname’ is specified: a list with the forcing function data sets, each present as a two-columned matrix, with (time,value); interpolation outside the interval [min(times), max(times)] is done by taking the value at the closest data extreme.
See forcings or package vignette "compiledCode".

initforc if not NULL, the name of the forcing function initialisation function, as provided in ‘dllname’. It MUST be present if forcings has been given a value. See forcings or package vignette "compiledCode".

fcontrol A list of control parameters for the forcing functions. See forcings or vignette compiledCode.

events A matrix or data frame that specifies events, i.e. when the value of a state variable is suddenly changed. See events for more information.

lags A list that specifies timelags, i.e. the number of steps that has to be kept. To be used for delay differential equations. See timelags, dede for more information.

... additional arguments passed to func and jacfunc allowing this to be a generic function.

Details

The work is done by the FORTRAN subroutine lsodar, whose documentation should be consulted for details (it is included as comments in the source file ‘src/opkdmain.f’). The implementation is based on the November, 2003 version of lsodar, from Netlib.

lsodar switches automatically between stiff and nonstiff methods (similar as lsoda). This means that the user does not have to determine whether the problem is stiff or not, and the solver will automatically choose the appropriate method. It always starts with the nonstiff method.

lsodar can find the root of at least one of a set of constraint functions rootfunc of the independent and dependent variables. It then returns the solution at the root if that occurs sooner than the specified stop condition, and otherwise returns the solution according the specified stop condition.
Caution: Because of numerical errors in the function rootfun due to roundoff and integration error, lsodar may return false roots, or return the same root at two or more nearly equal values of time.

The form of the Jacobian can be specified by jactype which can take the following values:

\textbf{jactype = "fullint"} \quad a full Jacobian, calculated internally by lsodar, the default,
\textbf{jactype = "fullusr"}: a full Jacobian, specified by user function jacfunc,
\textbf{jactype = "bandusr"}: a banded Jacobian, specified by user function jacfunc; the size of the bands specified by bandup and banddown,
\textbf{jactype = "bandint"}: banded Jacobian, calculated by lsodar; the size of the bands specified by bandup and banddown.

If jactype = "fullusr" or "bandusr" then the user must supply a subroutine jacfunc.

The input parameters rtol, and atol determine the error control performed by the solver. See lsoda for details.
The output will have the attribute `iroot`, if a root was found `iroot` is a vector, its length equal to the number of constraint functions it will have a value of 1 for the constraint function whose root that has been found and 0 otherwise.

The diagnostics of the integration can be printed to screen by calling `diagnostics`. If `verbose = TRUE`, the diagnostics will written to the screen at the end of the integration.

See vignette("deSolve") for an explanation of each element in the vectors containing the diagnostic properties and how to directly access them.

**Models** may be defined in compiled C or FORTRAN code, as well as in an R-function. See package vignette "compiledCode" for details.

More information about models defined in compiled code is in the package vignette ("compiled-Code"); information about linking forcing functions to compiled code is in forcings.

Examples in both C and FORTRAN are in the ‘dynload’ subdirectory of the deSolve package directory.

**Value**

A matrix of class `desolve` with up to as many rows as elements in `times` and as many columns as elements in `y` plus the number of "global" values returned in the next elements of the return from `func`, plus and additional column for the time value. There will be a row for each element in `times` unless the FORTRAN routine ‘lsodar’ returns with an unrecoverable error. If `y` has a names attribute, it will be used to label the columns of the output value.

If a root has been found, the output will have the attribute `iroot`, an integer indicating which root has been found.

**Author(s)**

Karline Soetaert <karline.soetaert@nioz.nl>

**References**


Netlib: [http://www.netlib.org](http://www.netlib.org)

**See Also**

- `roots` for more examples on roots and events
- `rk`, `rkMethod`, `rk4` and `euler` for Runge-Kutta integrators.
- `lsoda`, `lsode`, `lsodes`, `vode`, `daspk` for other solvers of the Livermore family,
- `ode` for a general interface to most of the ODE solvers,
- `ode.band` for solving models with a banded Jacobian,
- **ode.1D** for integrating 1-D models,
- **ode.2D** for integrating 2-D models,
- **ode.3D** for integrating 3-D models,
- **diagnostics** to print diagnostic messages.

**Examples**

```r
## Example 1:
## from lsodar source code

fun <- function(t, y, parms) {
  ydot <- vector(len = 3)
  return(list(ydot, ytot = sum(y)))
}

rootfun <- function(t, y, parms) {
  yroot <- vector(len = 2)
  return(yroot)
}

y <- c(1, 0, 0)
times <- c(0, 0.4*10^8)
Out <- NULL
ny <- length(y)

out <- lsodar(y = y, times = times, fun = Fun, rootfun = rootfun, rtol = 1e-4, atol = c(1e-6, 1e-10, 1e-6), parms = NULL)
print(paste("root is found for eqn", which(attributes(out)$iroot == 1)))
print(out[nrow(out),])
diagnostics(out)

## Example 2:
## using lsodar to estimate steady-state conditions

## Bacteria (Bac) are growing on a substrate (Sub)
model <- function(t, state, pars) {
  with(as.list(c(state, pars)), {
    ## substrate uptake death respiration
    dBact <- gmax*eff*Sub/(Sub+ks)*Bact - dB*Bact - rB*Bact
    dSub <- -gmax*Sub/(Sub+ks)*Bact + dB*Bact + input
    return(list(c(dBact, dSub)))
  })
```

```
lsode

Solver for Ordinary Differential Equations (ODE)

## root is the condition where sum of \( |\text{rates of change}| \)
## is very small

```r
code{
rootfun <- function(t, state, pars) {
  dstate <- unlist(model(t, state, pars)) # rate of change vector
  return(sum(abs(dstate)) - 1e-10)
}

tout <- c(0, 1e10)
state <- c(Bact = pars$Bini, Sub = pars$Sini)
out <- lsodar(state, tout, model, pars, rootfun = rootfun)
print(out)
}
```

## Example 3:
## using lsodar to trigger an event
## a state variable is decaying at a first-order rate.  
## when it reaches the value 0.1, a random amount is added.

```r
code{
derivfun <- function(t,y,pars)
  list (-0.05 * y)

dstate <- function(t, state, pars) # rate of change vector
  return(y - 0.1)

eventfun <- function(t,y,pars)
  return(y + runif(1))

yini <- 0.8
times <- 0:200

out <- lsodar(func=derivfun, y = yini, times=times,
  rootfun = rootfun, events = list(func=eventfun, root = TRUE))

plot(out, type = "l", lwd = 2, main = "lsodar with event")
```
Description

Solves the initial value problem for stiff or nonstiff systems of ordinary differential equations (ODE) in the form:

\[ \frac{dy}{dt} = f(t, y) \]

The \texttt{lsode} function provides an interface to the FORTRAN ODE solver of the same name, written by Alan C. Hindmarsh and Andrew H. Sherman.

It combines parts of the code \texttt{lsodar} and can thus find the root of at least one of a set of constraint functions \( g(i) \) of the independent and dependent variables. This can be used to stop the simulation or to trigger events, i.e. a sudden change in one of the state variables.

The system of ODE's is written as an \texttt{R} function or be defined in compiled code that has been dynamically loaded.

In contrast to \texttt{lsoda}, the user has to specify whether or not the problem is stiff and choose the appropriate solution method.

\texttt{lsode} is very similar to \texttt{vode}, but uses a fixed-step-interpolate method rather than the variable-coefficient method in \texttt{vode}. In addition, in \texttt{vode} it is possible to choose whether or not a copy of the Jacobian is saved for reuse in the corrector iteration algorithm; In \texttt{lsode}, a copy is not kept.

Usage

\texttt{lsode}(y, times, func, parms, rtol = 1e-6, atol = 1e-6, 
jacfunc = \texttt{NULL}, jactype = "fullint", mf = \texttt{NULL}, rootfunc = \texttt{NULL}, 
verbose = \texttt{FALSE}, nroot = 0, tcrit = \texttt{NULL}, hmin = 0, hmax = \texttt{NULL}, 
hini = 0, ynames = \texttt{TRUE}, maxord = \texttt{NULL}, bandup = \texttt{NULL}, banddown = \texttt{NULL}, 
maxsteps = 5000, dllname = \texttt{NULL}, initfunc = \texttt{dllname}, 
initpar = parms, rpar = \texttt{NULL}, ipar = \texttt{NULL}, nout = 0, 
outnames = \texttt{NULL}, forcings=\texttt{NULL}, initforc = \texttt{NULL}, 
fcontrol=\texttt{NULL}, events=\texttt{NULL}, lags = \texttt{NULL},...)

Arguments

\textit{y} the initial (state) values for the ODE system. If \textit{y} has a name attribute, the names will be used to label the output matrix.

\textit{times} time sequence for which output is wanted; the first value of \textit{times} must be the initial time; if only one step is to be taken; set \textit{times} = \texttt{NULL}.

\textit{func} either an \texttt{R}-function that computes the values of the derivatives in the ODE system (the \textit{model definition}) at time \textit{t}, or a character string giving the name of a compiled function in a dynamically loaded shared library.

If \textit{func} is an \texttt{R}-function, it must be defined as: \texttt{func <- function(t, y, parms,...).}

\textit{t} is the current time point in the integration, \textit{y} is the current estimate of the variables in the ODE system. If the initial values \textit{y} has a names attribute, the names will be available inside \textit{func}. \textit{parms} is a vector or list of parameters; ... (optional) are any other arguments passed to the function.

The return value of \textit{func} should be a list, whose first element is a vector containing the derivatives of \textit{y} with respect to \textit{time}, and whose next elements are
global values that are required at each point in times. The derivatives must be specified in the same order as the state variables \( y \).

If `func` is a string, then `dllname` must give the name of the shared library (without extension) which must be loaded before `lsode()` is called. See package vignette "compiledCode" for more details.

`parms` vector or list of parameters used in `func` or `jacfunc`.

`rtol` relative error tolerance, either a scalar or an array as long as \( y \). See details.

`atol` absolute error tolerance, either a scalar or an array as long as \( y \). See details.

`jacfunc` if not NULL, an R function that computes the Jacobian of the system of differential equations \( \partial \dot{y}_i / \partial y_j \), or a string giving the name of a function or subroutine in `dllname` that computes the Jacobian (see vignette "compiledCode" for more about this option).

In some circumstances, supplying `jacfunc` can speed up the computations, if the system is stiff. The R calling sequence for `jacfunc` is identical to that of `func`.

If the Jacobian is a full matrix, `jacfunc` should return a matrix \( \partial \dot{y}_i / \partial y_j \), where the \( i \)th row contains the derivative of \( \dot{y}_i \) with respect to \( y_j \), or a vector containing the matrix elements by columns (the way R and FORTRAN store matrices). If the Jacobian is banded, `jacfunc` should return a matrix containing only the nonzero bands of the Jacobian, rotated row-wise. See first example of `lsode`.

`jactype` the structure of the Jacobian, one of "fullint", "fullusr", "bandusr" or "bandint" - either full or banded and estimated internally or by user; overruled if `mf` is not NULL.

`mf` the "method flag" passed to function `lsode` - overrules `jactype` - provides more options than `jactype` - see details.

`rootfunc` if not NULL, an R function that computes the function whose root has to be estimated or a string giving the name of a function or subroutine in `dllname` that computes the root function. The R calling sequence for `rootfunc` is identical to that of `func`. `rootfunc` should return a vector with the function values whose root is sought.

`verbose` if TRUE: full output to the screen, e.g. will print the diagnostics of the integration - see details.

`nroot` only used if `dllname` is specified: the number of constraint functions whose roots are desired during the integration; if `rootfunc` is an R-function, the solver estimates the number of roots.

`tcrit` if not NULL, then `lsode` cannot integrate past `tcrit`. The FORTRAN routine `lsode` overshoots its targets (times points in the vector `times`), and interpolates values for the desired time points. If there is a time beyond which integration should not proceed (perhaps because of a singularity), that should be provided in `tcrit`.

`hmin` an optional minimum value of the integration stepsize. In special situations this parameter may speed up computations with the cost of precision. Don’t use `hmin` if you don’t know why!

`hmax` an optional maximum value of the integration stepsize. If not specified, `hmax` is set to the largest difference in `times`, to avoid that the simulation possibly ignores short-term events. If 0, no maximal size is specified.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hin</td>
<td>initial step size to be attempted; if 0, the initial step size is determined by the solver.</td>
</tr>
<tr>
<td>ynames</td>
<td>logical, if FALSE names of state variables are not passed to function func; this may speed up the simulation especially for multi-D models.</td>
</tr>
<tr>
<td>maxord</td>
<td>the maximum order to be allowed. NULL uses the default, i.e. order 12 if implicit Adams method (meth = 1), order 5 if BDF method (meth = 2). Reduce maxord to save storage space.</td>
</tr>
<tr>
<td>bandup</td>
<td>number of non-zero bands above the diagonal, in case the Jacobian is banded.</td>
</tr>
<tr>
<td>banddown</td>
<td>number of non-zero bands below the diagonal, in case the Jacobian is banded.</td>
</tr>
<tr>
<td>maxsteps</td>
<td>maximal number of steps per output interval taken by the solver.</td>
</tr>
<tr>
<td>dllname</td>
<td>a string giving the name of the shared library (without extension) that contains all the compiled function or subroutine definitions refered to in func and jacfunc. See package vignette compiledCode.</td>
</tr>
<tr>
<td>initfunc</td>
<td>if not NULL, the name of the initialisation function (which initialises values of parameters), as provided in ‘dllname’. See package vignette compiledCode.</td>
</tr>
<tr>
<td>initpar</td>
<td>only when ‘dllname’ is specified and an initialisation function initfunc is in the dll: the parameters passed to the initialiser, to initialise the common blocks (FORTRAN) or global variables (C, C++).</td>
</tr>
<tr>
<td>rpar</td>
<td>only when ‘dllname’ is specified: a vector with double precision values passed to the dll-functions whose names are specified by func and jacfunc.</td>
</tr>
<tr>
<td>ipar</td>
<td>only when ‘dllname’ is specified: a vector with integer values passed to the dll-functions whose names are specified by func and jacfunc.</td>
</tr>
<tr>
<td>nout</td>
<td>only used if dllname is specified and the model is defined in compiled code: the number of output variables calculated in the compiled function func, present in the shared library. Note: it is not automatically checked whether this is indeed the number of output variables calculated in the dll - you have to perform this check in the code - See package vignette compiledCode.</td>
</tr>
<tr>
<td>outnames</td>
<td>only used if ‘dllname’ is specified and nout &gt; 0: the names of output variables calculated in the compiled function func, present in the shared library. These names will be used to label the output matrix.</td>
</tr>
<tr>
<td>forcings</td>
<td>only used if ‘dllname’ is specified: a list with the forcing function data sets, each present as a two-columned matrix, with (time,value); interpolation outside the interval [min(times), max(times)] is done by taking the value at the closest data extreme. See forcings or package vignette compiledCode.</td>
</tr>
<tr>
<td>initforc</td>
<td>if not NULL, the name of the forcing function initialisation function, as provided in ‘dllname’. It MUST be present if forcings has been given a value. See forcings or package vignette compiledCode.</td>
</tr>
<tr>
<td>fcontrol</td>
<td>A list of control parameters for the forcing functions. See forcings or vignette compiledCode.</td>
</tr>
<tr>
<td>events</td>
<td>A matrix or data frame that specifies events, i.e. when the value of a state variable is suddenly changed. See events for more information.</td>
</tr>
<tr>
<td>lags</td>
<td>A list that specifies timelags, i.e. the number of steps that has to be kept. To be used for delay differential equations. See timelags, dede for more information.</td>
</tr>
</tbody>
</table>
additional arguments passed to `func` and `jacfunc` allowing this to be a generic function.

**Details**

The work is done by the FORTRAN subroutine `lsode`, whose documentation should be consulted for details (it is included as comments in the source file `src/opkdmain.f`). The implementation is based on the November, 2003 version of `lsode`, from Netlib.

Before using the integrator `lsode`, the user has to decide whether or not the problem is stiff.

If the problem is nonstiff, use method flag `mf = 10`, which selects a nonstiff (Adams) method, no Jacobian used.

If the problem is stiff, there are four standard choices which can be specified with `jactype` or `mf`.

The options for `jactype` are

- `jactype = "fullint"` a full Jacobian, calculated internally by `lsode`, corresponds to `mf = 22`,
- `jactype = "fullusr"` a full Jacobian, specified by user function `jacfunc`, corresponds to `mf = 21`,
- `jactype = "bandusr"` a banded Jacobian, specified by user function `jacfunc`; the size of the bands specified by `bandup` and `banddown`, corresponds to `mf = 24`,
- `jactype = "bandint"` a banded Jacobian, calculated by `lsode`; the size of the bands specified by `bandup` and `banddown`, corresponds to `mf = 25`.

More options are available when specifying `mf` directly.

The legal values of `mf` are 10, 11, 12, 13, 14, 15, 20, 21, 22, 23, 24, 25.

`mf` is a positive two-digit integer, `mf = (10*METH + MITER)`, where

- `METH` indicates the basic linear multistep method: `METH = 1` means the implicit Adams method.
- `METH = 2` means the method based on backward differentiation formulas (BDF-s).

- `MITER` indicates the corrector iteration method: `MITER = 0` means functional iteration (no Jacobian matrix is involved). `MITER = 1` means chord iteration with a user-supplied full (NEQ by NEQ) Jacobian. `MITER = 2` means chord iteration with an internally generated (difference quotient) full Jacobian (using NEQ extra calls to `func` per df/dy value). `MITER = 3` means chord iteration with an internally generated diagonal Jacobian approximation (using 1 extra call to `func` per df/dy evaluation). `MITER = 4` means chord iteration with a user-supplied banded Jacobian. `MITER = 5` means chord iteration with an internally generated banded Jacobian (using ML+MU+1 extra calls to `func` per df/dy evaluation).

If `MITER = 1` or 4, the user must supply a subroutine `jacfunc`.

Inspection of the example below shows how to specify both a banded and full Jacobian.

The input parameters `rtol`, and `atol` determine the error control performed by the solver. See `lsoda` for details.

The diagnostics of the integration can be printed to screen by calling `diagnostics`. If `verbose = TRUE`, the diagnostics will written to the screen at the end of the integration.

See vignette("deSolve") for an explanation of each element in the vectors containing the diagnostic properties and how to directly access them.

**Models** may be defined in compiled C or FORTRAN code, as well as in an R-function. See package vignette "compiledCode" for details.
More information about models defined in compiled code is in the package vignette ("compiled-Code"); information about linking forcing functions to compiled code is in forcings. Examples in both C and FORTRAN are in the ‘dynload’ subdirectory of the desolve package directory.

lsode can find the root of at least one of a set of constraint functions rootfunc of the independent and dependent variables. It then returns the solution at the root if that occurs sooner than the specified stop condition, and otherwise returns the solution according the specified stop condition.

Caution: Because of numerical errors in the function rootfun due to roundoff and integration error, lsode may return false roots, or return the same root at two or more nearly equal values of time.

Value

A matrix of class desolve with up to as many rows as elements in times and as many columns as elements in y plus the number of "global" values returned in the next elements of the return from func, plus and additional column for the time value. There will be a row for each element in times unless the FORTRAN routine ‘lsode’ returns with an unrecoverable error. If y has a names attribute, it will be used to label the columns of the output value.

Author(s)

Karline Soetaert <karline.soetaert@nioz.nl>

References


See Also

• rk,
• rk4 and euler for Runge-Kutta integrators.
• lsoda, lsodes, lsodar, vode, daspk for other solvers of the Livermore family,
• ode for a general interface to most of the ODE solvers,
• ode.band for solving models with a banded Jacobian,
• ode.1D for integrating 1-D models,
• ode.2D for integrating 2-D models,
• ode.3D for integrating 3-D models,

diagnostics to print diagnostic messages.

Examples

```r
## Equivalent to: # more information
```

```r
## Example 1:
## Various ways to solve the same model.
```

```r
## the model, 5 state variables
```
f1 <- function (t, y, parms) {
  ydot <- vector(len = 5)
  return(list(ydot))
}

## the Jacobian, written as a full matrix
fulljac <- function (t, y, parms) {
  jac <- matrix(nrow = 5, ncol = 5, byrow = TRUE,
                data = c(0.1, -0.2, 0, 0, 0,
                         -0.3, 0, -0.2, 0, 0,
                         0, -0.3, 0, -0.2, 0,
                         0, 0, -0.3, 0, -0.2,
                         0, 0, 0, -0.3, 0.1))
  return(jac)
}

## the Jacobian, written in banded form
bandjac <- function (t, y, parms) {
  jac <- matrix(nrow = 3, ncol = 5, byrow = TRUE,
                data = c(0, -0.2, -0.2, -0.2, -0.2,
                         0.1, 0.1, 0.1, 0.1, 0.1,
                         -0.3, -0.3, -0.3, 0))
  return(jac)
}

## initial conditions and output times
yini <- 1:5
times <- 1:20

## default: stiff method, internally generated, full Jacobian
out1 <- lsode(yini, times, f1, parms = 0, jactype = "fullint")

## stiff method, user-generated full Jacobian
out2 <- lsode(yini, times, f1, parms = 0, jactype = "fullusr",
              jacfunc = fulljac)

## stiff method, internally-generated banded Jacobian
## one nonzero band above (up) and below(down) the diagonal
out3 <- lsode(yini, times, f1, parms = 0, jactype = "bandint",
              bandup = 1, banddown = 1)

## stiff method, user-generated banded Jacobian
out4 <- lsode(yini, times, f1, parms = 0, jactype = "bandusr",
              jacfunc = bandjac, bandup = 1, banddown = 1)

## non-stiff method
out5 <- lsode(yini, times, f1, parms = 0, mf = 10)

## Example 2:
## diffusion on a 2-D grid
## partially specified Jacobian
##

diffusion2D <- function(t, Y, par) {
  y <- matrix(nrow = n, ncol = n, data = Y)
  dy <- r*y   # production

  ## diffusion in X-direction; boundaries = 0-concentration
  Flux <- -Dx * rbind(y[1,],[y[2:n,]-y[1:(n-1),]],-y[n,])/dx
  dY <- dY - (Flux[2:(n+1),]-Flux[1:n,])/dx

  ## diffusion in Y-direction
  Flux <- -Dy * cbind(y[,1],[y[,2:n]-y[1:(n-1)],]-y[,n])/dy
  dY <- dY - (Flux[2:(n+1)]-Flux[,1:n])/dy

  return(list(as.vector(dY)))
}

## parameters
dy <- dx <- 1    # grid size
Dy <- Dx <- 1    # diffusion coeff, X- and Y-direction
r    <- 0.025    # production rate

n <- 50
y <- matrix(nrow = n, ncol = n, 0)

pa <- par(ask = FALSE)

## initial condition
for (i in 1:n) {
  for (j in 1:n) {
    dst <- (i - n/2)^2 + (j - n/2)^2
    y[i, j] <- max(0, 1 - 1/(n*n) * (dst - n)^2)
  }
}

filled.contour(y, color.palettr = terrain.colors)

## jacfunc need not be estimated exactly
## a crude approximation, with a smaller bandwidth will do.
## Here the half-bandwidth 1 is used, whereas the true
## half-bandwidths are equal to n.
## This corresponds to ignoring the y-direction coupling in the ODEs.

print(system.time{
  for (i in 1:20) {
    ...}})
lsodes

Solver for Ordinary Differential Equations (ODE) With Sparse Jacobian

Description

Solves the initial value problem for stiff systems of ordinary differential equations (ODE) in the form:

$$ \frac{dy}{dt} = f(t, y) $$

and where the Jacobian matrix df/dy has an arbitrary sparse structure.

The R function lsodes provides an interface to the FORTRAN ODE solver of the same name, written by Alan C. Hindmarsh and Andrew H. Sherman.

The system of ODE's is written as an R function or be defined in compiled code that has been dynamically loaded.

Usage

```
lsodes(y, times, func, parms, rtol = 1e-6, atol = 1e-6,
      jacvec = NULL, sparsetype = "sparseint", nnz = NULL,
      inz = NULL, rootfunc = NULL,
      verbose = FALSE, nroot = 0, tcrit = NULL, hmin = 0,
      hmax = NULL, hini = 0, ynames = TRUE, maxord = NULL,
      maxsteps = 5000, lrw = NULL, liw = NULL, dllname = NULL,
      initfunc = dllname, initpar = parms, rpar = NULL,
      ipar = NULL, nout = 0, outnames = NULL, forcings=NULL,
      initforc = NULL, fcontrol=NULL, events=NULL, lags = NULL,
      ...)```

Arguments

- `y` the initial (state) values for the ODE system. If `y` has a name attribute, the names will be used to label the output matrix.
- `times` time sequence for which output is wanted; the first value of `times` must be the initial time; if only one step is to be taken; set `times = NULL`. 

```r
out <- lsode(func = diffusion2D, y = as.vector(y), times = times,
              parms = NULL, jactype = "bandint", bandup = 1, banddown = 1)
filled.contour(matrix(nrow = n, ncol = n, out[2,-1]), zlim = c(0,1),
              color.palette = terrain.colors, main = i)

y <- out[2, -1]
)
```
func either an R-function that computes the values of the derivatives in the ODE system (the model definition) at time $t$, or a character string giving the name of a compiled function in a dynamically loaded shared library.

If `func` is an R-function, it must be defined as: `func <- function(t, y, parms, ...).` $t$ is the current time point in the integration, $y$ is the current estimate of the variables in the ODE system. If the initial values $y$ has a names attribute, the names will be available inside `func`. `parms` is a vector or list of parameters; … (optional) are any other arguments passed to the function.

The return value of `func` should be a list, whose first element is a vector containing the derivatives of $y$ with respect to time, and whose next elements are global values that are required at each point in `times`. The derivatives must be specified in the same order as the state variables $y$.

If `func` is a string, then `dllname` must give the name of the shared library (without extension) which must be loaded before `lsodes()` is called. See package vignette "compiledCode" for more details.

parms vector or list of parameters used in `func` or `jacfunc`.

rtol relative error tolerance, either a scalar or an array as long as `y`. See details.

atol absolute error tolerance, either a scalar or an array as long as `y`. See details.

jacvec if not NULL, an R function that computes a column of the Jacobian of the system of differential equations $\partial y_i/\partial y_j$, or a string giving the name of a function or subroutine in 'dllname' that computes the column of the Jacobian (see vignette "compiledCode" for more about this option).

The R calling sequence for jacvec is identical to that of `func`, but with extra parameter `j`, denoting the column number. Thus, jacvec should be called as: `jacvec = func(t, y, j, parms)` and jacvec should return a vector containing column `j` of the Jacobian, i.e. its i-th value is $\partial y_i/\partial y_j$. If this function is absent, `lsodes` will generate the Jacobian by differences.

sparsetype the sparsity structure of the Jacobian, one of "sparseint" or "sparseusr", "sparsejan", ... The sparsity can be estimated internally by `lsodes` (first option) or given by the user (last two). See details.

nnz the number of nonzero elements in the sparse Jacobian (if this is unknown, use an estimate).

inz if `sparsetype` equal to "sparseusr", a two-columned matrix with the (row, column) indices to the nonzero elements in the sparse Jacobian. If `sparsetype` = "sparsejan", a vector with the elements `ian` followed by the elements `jan` as used in the `lsodes` code. See details. In all other cases, ignored.

rootfunc if not NULL, an R function that computes the function whose root has to be estimated or a string giving the name of a function or subroutine in 'dllname' that computes the root function. The R calling sequence for rootfunc is identical to that of `func`. rootfunc should return a vector with the function values whose root is sought.

verbose if TRUE: full output to the screen, e.g. will print the diagnostics of the integration - see details.

nroot only used if 'dllname' is specified: the number of constraint functions whose roots are desired during the integration; if rootfunc is an R-function, the solver estimates the number of roots.
lsodes

if not NULL, then lsodes cannot integrate past t

The FORTRAN routine lsodes overshoots its targets (times points in the vector times), and interpolates values for the desired time points. If there is a time beyond which integration should not proceed (perhaps because of a singularity), that should be provided in tcrit.

hmin

an optional minimum value of the integration steps size. In special situations this parameter may speed up computations with the cost of precision. Don't use hmin if you don't know why!

hmax

an optional maximum value of the integration steps size. If not specified, hmax is set to the largest difference in times, to avoid that the simulation possibly ignores short-term events. If 0, no maximal size is specified.

hini

initial step size to be attempted; if 0, the initial step size is determined by the solver.

ynames

logical, if FALSE names of state variables are not passed to function func; this may speed up the simulation especially for multi-D models.

maxord

the maximum order to be allowed. NULL uses the default, i.e. order 12 if implicit Adams method (meth = 1), order 5 if BDF method (meth = 2). Reduce maxord to save storage space.

maxsteps

maximal number of steps per output interval taken by the solver.

lrw

the length of the real work array rwork; due to the sparsity, this cannot be readily predicted. If NULL, a guess will be made, and if not sufficient, lsodes will return with a message indicating the size of rwork actually required. Therefore, some experimentation may be necessary to estimate the value of lrw.

For instance, if you get the error:

DLSODES- RWORK length is insufficient to proceed.

Length needed is .ge. LENRW (=I1), exceeds LRW (=I2)

In above message, I1 = 27627 I2 = 25932

set lrw equal to 27627 or a higher value.

liw

the length of the integer work array iwork; due to the sparsity, this cannot be readily predicted. If NULL, a guess will be made, and if not sufficient, lsodes will return with a message indicating the size of iwork actually required. Therefore, some experimentation may be necessary to estimate the value of liw.

dllname

a string giving the name of the shared library (without extension) that contains all the compiled function or subroutine definitions referred to in func and jacfunc. See package vignette "compiledCode".

initfunc

if not NULL, the name of the initialisation function (which initialises values of parameters), as provided in 'dllname'. See package vignette "compiledCode".

initpar

only when 'dllname' is specified and an initialisation function initfunc is in the dll: the parameters passed to the initialiser, to initialise the common blocks (FORTRAN) or global variables (C, C++).

rpar

only when 'dllname' is specified: a vector with double precision values passed to the dll-functions whose names are specified by func and jacfunc.

ipar

only when 'dllname' is specified: a vector with integer values passed to the dll-functions whose names are specified by func and jacfunc.
nout

only used if dllname is specified and the model is defined in compiled code: the number of output variables calculated in the compiled function func, present in the shared library. Note: it is not automatically checked whether this is indeed the number of output variables calculated in the dll - you have to perform this check in the code. See package vignette "compiledCode".

outnames

only used if 'dllname' is specified and nout > 0: the names of output variables calculated in the compiled function func, present in the shared library. These names will be used to label the output matrix.

forcings

only used if 'dllname' is specified: a list with the forcing function data sets, each present as a two-columned matrix, with (time,value); interpolation outside the interval [min(times), max(times)] is done by taking the value at the closest data extreme.

See forcings or package vignette "compiledCode".

initforc

if not NULL, the name of the forcing function initialisation function, as provided in 'dllname'. It MUST be present if forcings has been given a value. See forcings or package vignette "compiledCode".

fcontrol

A list of control parameters for the forcing functions. See forcings or vignette compiledCode.

events

A matrix or data frame that specifies events, i.e. when the value of a state variable is suddenly changed. See events for more information.

lags

A list that specifies timelags, i.e. the number of steps that has to be kept. To be used for delay differential equations. See timelags, dede for more information.

Additional arguments passed to func and jacfunc allowing this to be a generic function.

Details

The work is done by the FORTRAN subroutine lsodes, whose documentation should be consulted for details (it is included as comments in the source file ‘src/opkdmain.f’). The implementation is based on the November, 2003 version of lsodes, from Netlib.

lsodes is applied for stiff problems, where the Jacobian has a sparse structure.

There are several choices depending on whether jacvec is specified and depending on the setting of sparsetype.

If function jacvec is present, then it should return the j-th column of the Jacobian matrix.

There are also several choices for the sparsity specification, selected by argument sparsetype.

- sparsetype = "sparseint". The sparsity is estimated by the solver, based on numerical differences. In this case, it is advisable to provide an estimate of the number of non-zero elements in the Jacobian (nnz). This value can be approximate; upon return the number of nonzero elements actually required will be known (1st element of attribute dims). In this case, inz need not be specified.

- sparsetype = "sparseusr". The sparsity is determined by the user. In this case, inz should be a matrix, containing indices (row, column) to the nonzero elements in the Jacobian matrix. The number of nonzeros nnz will be set equal to the number of rows in inz.
lsodes

- \texttt{sparsetype = "sparsejan"}. The sparsity is also determined by the user. In this case, \texttt{inz} should be a vector, containing the \texttt{ian} and \texttt{jan} elements of the sparse storage format, as used in the sparse solver. Elements of \texttt{ian} should be the first \(n+1\) elements of this vector, and contain the starting locations in \texttt{jan} of columns 1.. \(n\). \texttt{jan} contains the row indices of the nonzero locations of the Jacobian, reading in columnwise order. The number of nonzeros \(nnz\) will be set equal to the length of \texttt{inz} - (\(n+1\)).

- \texttt{sparsetype = "1D", "2D", "3D"}. The sparsity is estimated by the solver, based on numerical differences. Assumes finite differences in a 1D, 2D or 3D regular grid - used by functions \texttt{ode.1d}, \texttt{ode.2d}, \texttt{ode.3d}. Similar are \texttt{2Dmap"}, and \texttt{3Dmap"}, which also include a mapping variable (passed in \texttt{nnz}).

The input parameters \texttt{rtol}, and \texttt{atol} determine the \textbf{error control} performed by the solver. See \texttt{lsoda} for details.

The diagnostics of the integration can be printed to screen by calling \texttt{diagnostics}. If \texttt{verbose = TRUE}, the diagnostics will written to the screen at the end of the integration.

See vignette("deSolve") for an explanation of each element in the vectors containing the diagnostic properties and how to directly access them.

**Models** may be defined in compiled C or FORTRAN code, as well as in an R-function. See package vignette "compiledCode" for details.

More information about models defined in compiled code is in the package vignette ("compiledCode"); information about linking forcing functions to compiled code is in \texttt{forcings}.

Examples in both C and FORTRAN are in the ‘doc/examples/dynload’ subdirectory of the deSolve package directory.

\texttt{lsodes} can find the root of at least one of a set of constraint functions \texttt{rootfunc} of the independent and dependent variables. It then returns the solution at the root if that occurs sooner than the specified stop condition, and otherwise returns the solution according the specified stop condition.

Caution: Because of numerical errors in the function \texttt{rootfun} due to roundoff and integration error, \texttt{lsodes} may return false roots, or return the same root at two or more nearly equal values of \texttt{time}.

**Value**

A matrix of class \texttt{desolve} with up to as many rows as elements in \texttt{times} and as many columns as elements in \texttt{y} plus the number of "global" values returned in the next elements of the return from \texttt{func}, plus and additional column for the time value. There will be a row for each element in \texttt{times} unless the FORTRAN routine \texttt{lsodes} returns with an unrecoverable error. If \texttt{y} has a names attribute, it will be used to label the columns of the output value.

**Author(s)**

Karline Soetaert <karline.soetaert@nioz.nl>

**References**


See Also

- \texttt{rk},
- \texttt{rk4} and \texttt{euler} for Runge-Kutta integrators.
- \texttt{lsoda}, \texttt{lsode}, \texttt{lsodar}, \texttt{vode}, \texttt{daspk} for other solvers of the Livermore family,
- \texttt{ode} for a general interface to most of the ODE solvers,
- \texttt{ode.band} for solving models with a banded Jacobian,
- \texttt{ode.1D} for integrating 1-D models,
- \texttt{ode.2D} for integrating 2-D models,
- \texttt{ode.3D} for integrating 3-D models,
- \texttt{diagnostics} to print diagnostic messages.

Examples

```r
## Various ways to solve the same model.
##
## The example from lsodes source code
## A chemical model
##
##
# n <- 12
# y <- rep(1, n)
# dy <- rep(0, n)
# times <- c(0, 0.1*(10*(0:4)))
# rtol <- 1.0e-4
# atol <- 1.0e-6
#
# parms <- c(rk1 = 0.1, rk2 = 10.0, rk3 = 50.0, rk4 = 2.5, rk5 = 0.1,
#            rk6 = 10.0, rk7 = 50.0, rk8 = 2.5, rk9 = 50.0, rk10 = 5.0,
#            rk11 = 50.0, rk12 = 50.0, rk13 = 50.0, rk14 = 30.0,
#            rk15 = 100.0, rk16 = 2.5, rk17 = 100.0, rk18 = 2.5,
#            rk19 = 50.0, rk20 = 50.0)
#
# chemistry <- function (time, Y, pars) {
#   with (as.list(pars), {
#     dy[1] <- -rk1 *Y[1]
#   })
```

Isodes
lsodes

                 rk6  *Y[10]  - rk9  *Y[10]
    return(list(dy))
}

## application 1. lsodes estimates the structure of the Jacobian
## and calculates the Jacobian by differences
## 
out <- lsodes(func = chemistry, y = y, parms = parms, times = times,
         atol = atol, rtol = rtol, verbose = TRUE)

## application 2. the structure of the Jacobian is input
## lsodes calculates the Jacobian by differences
## this is not so efficient...
##
## elements of Jacobian that are not zero
nonzero <- matrix(nc = 2, byrow = TRUE, data = c(
    1, 1, 2, 1,  # influence of sp1 on rate of change of others
    2, 2, 3, 2, 4, 2, 5, 2, 12, 2,
    2, 3, 3, 3, 4, 3, 6, 3, 10, 3,
    2, 4, 3, 4, 4, 4, 9, 4,  # d (dy)/dy4
    2, 5, 5, 5, 9, 5, 12, 5,
    3, 6, 6, 6, 9, 6, 10, 6,
    7, 7, 9, 7, 10, 7, 12, 7,
    8, 8, 10, 8, 11, 8,
    3,10, 6,10, 7,10, 8,10, 10,10, 12,10,
    2,12, 5,12, 7,12, 10,12, 12,12)
)

## when run, the default length of rwork is too small
## lsodes will tell the length actually needed
# out2 <- lsodes(func = chemistry, y = y, parms = parms, times = times,
#               inz = nonzero, atol = atol, rtol = rtol)  # gives warning
out2 <- lsodes(func = chemistry, y = y, parms = parms, times = times,
               spartype = "sparseusr", inz = nonzero,
               atol = atol, rtol = rtol, verbose = TRUE, lrw = 353)

## application 3. lsodes estimates the structure of the Jacobian
## the Jacobian (vector) function is input
## 箸paragus ---------------------------------------------------------------

```
chemjac <- function (time, Y, j, pars) {
  with (as.list(pars), {
    PDJ <- rep(8, n)

    if (j == 1) {
      PDJ[1] <- -rk1
      PDJ[2] <- -rk1
    } else if (j == 2) {
      PDJ[12] <- -rk15 * Y[12]
    } else if (j == 3) {
      PDJ[10] <- rk5 - rk7 * Y[10]
    } else if (j == 4) {
      PDJ[2] <- rk11 * rk14
      PDJ[4] <- -rk11 * rk14 - rk4
      PDJ[9] <- rk4
    } else if (j == 5) {
      PDJ[2] <- rk19 * rk14
      PDJ[5] <- -rk19 * rk14 - rk16
      PDJ[9] <- rk16
      PDJ[12] <- -rk19 * rk14
    } else if (j == 6) {
      PDJ[3] <- rk12 * rk14
      PDJ[6] <- -rk12 * rk14 - rk8
      PDJ[9] <- rk8
      PDJ[10] <- rk12 * rk14
    } else if (j == 7) {
      PDJ[7] <- -rk20 * rk14 - rk18
      PDJ[9] <- rk18
      PDJ[10] <- -rk20 * rk14
      PDJ[12] <- -rk20 * rk14
    } else if (j == 8) {
      PDJ[8] <- -rk13 * rk14 - rk10
      PDJ[10] <- rk13 * rk14
    } else if (j == 10) {
      PDJ[8] <- rk9
      PDJ[12] <- rk6 - rk17 * Y[12]
    } else if (j == 12) {
```

ode

General Solver for Ordinary Differential Equations

Description

Solves a system of ordinary differential equations; a wrapper around the implemented ODE solvers

Usage


## S3 method for class 'deSolve'
print(x, ...)

## S3 method for class 'deSolve'
summary(object, select = NULL, which = select, subset = NULL, ...)
Arguments

\( y \)
the initial (state) values for the ODE system, a vector. If \( y \) has a name attribute, the names will be used to label the output matrix.

\( \text{times} \)
time sequence for which output is wanted; the first value of \( \text{times} \) must be the initial time.

\( \text{func} \)
either an \( \mathbb{R} \)-function that computes the values of the derivatives in the ODE system (the model definition) at time \( t \), or a character string giving the name of a compiled function in a dynamically loaded shared library.

If \( \text{func} \) is an \( \mathbb{R} \)-function, it must be defined as: \( \text{func} \left< \text{function} \left( t, y, \text{parms}, \ldots \right) \right. \). \( t \) is the current time point in the integration, \( y \) is the current estimate of the variables in the ODE system. If the initial values \( y \) has a names attribute, the names will be available inside \( \text{func} \). \( \text{parms} \) is a vector or list of parameters; \( \ldots \) (optional) are any other arguments passed to the function.

The return value of \( \text{func} \) should be a list, whose first element is a vector containing the derivatives of \( y \) with respect to time, and whose next elements are global values that are required at each point in \( \text{times} \). The derivatives must be specified in the same order as the state variables \( y \).

If \( \text{func} \) is a string, then \( \text{dllname} \) must give the name of the shared library (without extension) which must be loaded before \( \text{ode} \) is called. See package vignette "compiledCode" for more details.

\( \text{parms} \)
parameters passed to \( \text{func} \).

\( \text{method} \)
the integrator to use, either a \texttt{function} that performs integration, or a \texttt{list} of class \texttt{rkMethod}, or a \texttt{string} ("lsoda", "lsode", "lsodes", "lsodar", "vode", "daspk", "euler", "rk4", "ode23", "ode45", "radua", "bdf", "bdf_d", "adams", "impAdams" or "impAdams_d", "iteration"). Options "bdf", "bdf_d", "adams", "impAdams" or "impAdams_d" are the backward differentiation formula, the BDF with diagonal representation of the Jacobian, the (explicit) Adams and the implicit Adams method, and the implicit Adams method with diagonal representation of the Jacobian respectively (see details). The default integrator used is \texttt{lsoda}.

Method "iteration" is special in that here the function \( \text{func} \) should return the new value of the state variables rather than the rate of change. This can be used for individual based models, for difference equations, or in those cases where the integration is performed within \( \text{func} \). See last example.

\( \text{x} \)
an object of class \texttt{deSolve}, as returned by the integrators, and to be printed or to be subsetted.

\( \text{object} \)
an object of class \texttt{deSolve}, as returned by the integrators, and whose summary is to be calculated. In contrast to R's default, this returns a \texttt{data.frame}. It returns one summary column for a multi-dimensional variable.

\( \text{which} \)
the name(s) or the index to the variables whose summary should be estimated. Default = all variables.

\( \text{select} \)
which variable/columns to be selected.

\( \text{subset} \)
logical expression indicating elements or rows to keep when calculating a summary: missing values are taken as \texttt{FALSE}.

\( \ldots \)
additional arguments passed to the integrator or to the methods.
Details

This is simply a wrapper around the various ode solvers. See package vignette for information about specifying the model in compiled code. See the selected integrator for the additional options.

The default integrator used is `lsoda`. The option `method = "bdf"` provides a handle to the backward differentiation formula (it is equal to using `method = "lsode"`). It is best suited to solve stiff (systems of) equations.

The option `method = "bdf_d"` selects the backward differentiation formula that uses Jacobi-Newton iteration (neglecting the off-diagonal elements of the Jacobian (it is equal to using `method = "lsode", mf = 23`)). It is best suited to solve stiff (systems of) equations.

`method = "adams"` triggers the Adams method that uses functional iteration (no Jacobian used); (equal to `method = "lsode", mf = 10`). It is often the best choice for solving non-stiff (systems of) equations. Note: when functional iteration is used, the method is often said to be explicit, although it is in fact implicit.

`method = "impAdams"` selects the implicit Adams method that uses Newton-Raphson iteration (equal to `method = "lsode", mf = 12`).

`method = "impAdams_d"` selects the implicit Adams method that uses Jacobi-Newton iteration, i.e. neglecting all off-diagonal elements (equal to `method = "lsode", mf = 13`).

For very stiff systems, `method = "daspk"` may outperform `method = "bdf"`.

Value

A matrix of class `desolve` with up to as many rows as elements in `times` and as many columns as elements in `y` plus the number of "global" values returned in the second element of the return from `func`, plus an additional column (the first) for the time value. There will be one row for each element in `times` unless the integrator returns with an unrecoverable error. If `y` has a names attribute, it will be used to label the columns of the output value.

Author(s)

Karline Soetaert <karline.soetaert@nioz.nl>

See Also

- `plot.desolve` for plotting the outputs,
- `dede` general solver for delay differential equations
- `ode.band` for solving models with a banded Jacobian,
- `ode.1D` for integrating 1-D models,
- `ode.2D` for integrating 2-D models,
- `ode.3D` for integrating 3-D models,
- `aquaphy`, `ccl4model`, where ode is used,
- `lsoda`, `lsode`, `lsodes`, `lsodar`, `vode`, `daspk`, `radau`,
- `rk`, `rkMethod` for additional Runge-Kutta methods,
- `forcings` and `events`,
- `diagnostics` to print diagnostic messages.
Examples

```r
## Example 1: Predator-Prey Lotka-Volterra model (with logistic prey)
LVmod <- function(Time, State, Pars) {
  with(as.list(c(State, Pars)), {
    Ingestion <- rIng * Prey * Predator
    GrowthPrey <- rGrow * Prey * (1 - Prey/K)
    MortPredator <- rMort * Predator

    dPrey <- GrowthPrey - Ingestion
    dPredator <- Ingestion * assEff - MortPredator

    return(list(c(dPrey, dPredator)))
  })
}

pars <- c(rIng = 0.2, #/day, rate of ingestion
          rGrow = 1.0,  #/day, growth rate of prey
          rMort = 0.2,  #/day, mortality rate of predator
          assEff = 0.5,  # -, assimilation efficiency
          K = 10)       # mmol/m3, carrying capacity

yini <- c(Prey = 1, Predator = 2)
times <- seq(0, 200, by = 1)
out <- ode(yini, times, LVmod, pars)
summary(out)

## Default plot method
plot(out)

## User specified plotting
matplot(out[, 1], out[, 2:3], type = "l", xlab = "time", ylab = "Conc",
        main = "Lotka-Volterra", lwd = 2)
legend("topright", c("prey", "predator"), col = 1:2, lty = 1:2)
```

## Example 2: Substrate-Producer-Consumer Lotka-Volterra model

```r
SPCmod <- function(t, x, parms, input) {
  with(as.list(c(parms, x)), {
    import <- input(t)
    dS <- -import - b*S*P + g*C    # substrate
    dP <- c*S*P - d*C*P            # producer
    dC <- e*P*C - f*C             # consumer
  })
}
```r
res <- c(dS, dP, dC)
list(res)
}

## The parameters
deps <- c(b = 0.001, c = 0.1, d = 0.1, e = 0.1, f = 0.1, g = 0.0)

## vector of timesteps
times <- seq(0, 200, length = 101)

## external signal with rectangle impulse
signal <- data.frame(times = times,
                      import = rep(0, length(times)))

signal$import[signal$times >= 10 & signal$times <= 11] <- 0.2

## Start values for steady state
xstart <- c(S = 1, P = 1, C = 1)

## Solve model
out <- ode(y = xstart, times = times,
           func = SPMod, parms = deps, input = sigimp)

## Default plot method
plot(out)

## User specified plotting
mf <- par(mfrow = c(1, 2))
matplot(out[,1], out[,2:4], type = "l", xlab = "time", ylab = "state")
legend("topright", cex = 1.3, col = 1:3, lty = 1:3, legend = c("S", "P", "C"))
plot(out[,"P"], out[,"C"], type = "l", lwd = 2, xlab = "producer",
      ylab = "consumer")
par(mfrow = mf)

# Example 3: Discrete time model - using method = "iteration"
# The host-parasitoid model from Soetaert and Herman, 2009,
# Springer - p. 284.
Parasite <- function(t, y, ks) {
P <- y[1]
H <- y[2]
f <- A * P / (ks + H)
Pnew <- H * (1 - exp(-f))
Hnew <- H * exp(rH * (1 - H) - f)
list(c(Pnew, Hnew))
}
rH <- 2.82 # rate of increase
```
A <- 100  # attack rate
ks <- 15   # half-saturation density

out <- ode(func = Parasite, y = c(P = 0.5, H = 0.5), times = 0:50, parms = ks,
           method = "iteration")

out2<- ode(func = Parasite, y = c(P = 0.5, H = 0.5), times = 0:50, parms = 25,
            method = "iteration")

out3<- ode(func = Parasite, y = c(P = 0.5, H = 0.5), times = 0:50, parms = 35,
            method = "iteration")

## Plot all 3 scenarios in one figure
plot(out, out2, out3, lty = 1, lwd = 2)

## Same like "out", but *output* every two steps
## hini = 1 ensures that the same *internal* timestep of 1 is used
outb <- ode(func = Parasite, y = c(P = 0.5, H = 0.5),
             times = seq(0, 50, 2), hini = 1, parms = ks,
             method = "iteration")
plot(out, outb, type = c("l", "p"))

## Not run:
## ================================================================================
## Example4: Playing with the Jacobian options - see e.g. lsoda help page
##
## IMPORTANT: The following example is temporarily broken because of
## incompatibility with R 3.0 on some systems.
## A fix is on the way.
## ================================================================================

## a stiff equation, exponential decay, run 500 times
stiff <- function(t, y, p) {  # y and r are a 500-valued vector
  list(-r * y)
}

N <- 500
r <- runif(N, 15, 20)
yini <- runif(N, 1, 40)
times <- 0:10

## Using the default
print(system.time(
  out <- ode(y = yini, parms = NULL, times = times, func = stiff)
))
# diagnostics(out) shows that the method used = bdf (2), so it it stiff

## Specify that the Jacobian is banded, with nonzero values on the
## diagonal, i.e. the bandwidth up and down = 0
print(system.time(
  out2 <- ode(y = yini, parms = NULL, times = times, func = stiff,
ode.1D

```r
jactype = "bandint", bandup = 0, banddown = 0)
}

## Now we also specify the Jacobian function

jacob <- function(t, y, p) -r

print(system.time(
   out3 <- ode(y = yini, parms = NULL, times = times, func = stiff,
                jacfunc = jacob, jactype = "bandusr",
                bandup = 0, banddown = 0)
))

## The larger the value of N, the larger the time gain...

## End(Not run)
```

ode.1D

**Solver For Multicomponent 1-D Ordinary Differential Equations**

**Description**

Solves a system of ordinary differential equations resulting from 1-Dimensional partial differential equations that have been converted to ODEs by numerical differencing.

**Usage**

```r
ode.1D(y, times, func, parms, nspec = NULL, dimens = NULL,
       method = c("lsoda", "lsode", "lsodes", "lsodar", "vode", "daspk",
                   "euler", "rk4", "ode23", "ode45", "radua", "bdf", "adams", "impAdams",
                   "iteration"),
       names = NULL, bandwidth = 1, restructure = FALSE, ...)
```

**Arguments**

- `y` the initial (state) values for the ODE system, a vector. If `y` has a name attribute, the names will be used to label the output matrix.
- `times` time sequence for which output is wanted; the first value of `times` must be the initial time.
- `func` either an R-function that computes the values of the derivatives in the ODE system (the model definition) at time `t`, or a character string giving the name of a compiled function in a dynamically loaded shared library.
  
  If `func` is an R-function, it must be defined as: `func <- function(t, y, parms, ...)`. `t` is the current time point in the integration, `y` is the current estimate of the variables in the ODE system. If the initial values `y` has a names attribute, the names will be available inside `func`. `parms` is a vector or list of parameters; ... (optional) are any other arguments passed to the function.
  
  The return value of `func` should be a list, whose first element is a vector containing the derivatives of `y` with respect to time, and whose next elements are
ode.1D

global values that are required at each point in times. The derivatives must be specified in the same order as the state variables y.
If func is a character string then integrator lsodes will be used. See details.

parms parameters passed to func.

nspec the number of species (components) in the model. If NULL, then dimens should be specified.
dimens the number of boxes in the model. If NULL, then nspec should be specified.
method the integrator. Use "vode", "lsode", "lsoda", "lsodar", "daspk", or "lsodes" if the model is very stiff; "impAdams" or "radua" may be best suited for mildly stiff problems; "euler", "rk4", "ode23", "ode45", "adams" are most efficient for non-stiff problems. Also allowed is to pass an integrator function. Use one of the other Runge-Kutta methods via rkMethod. For instance, method = rkMethod("ode45ck") will trigger the Cash-Karp method of order 4(5).

Method "iteration" is special in that here the function func should return the new value of the state variables rather than the rate of change. This can be used for individual based models, for difference equations, or in those cases where the integration is performed within func)

names the names of the components; used for plotting.
bandwidth the number of adjacent boxes over which transport occurs. Normally equal to 1 (box i only interacts with box i-1, and i+1). Values larger than 1 will not work with method = "lsodes". Ignored if the method is explicit.
restructure whether or not the Jacobian should be restructured. Only used if the method is an integrator function. Should be TRUE if the method is implicit, FALSE if explicit.

... additional arguments passed to the integrator.

Details

This is the method of choice for multi-species 1-dimensional models, that are only subjected to transport between adjacent layers.

More specifically, this method is to be used if the state variables are arranged per species:

Two methods are implemented.

• The default method rearranges the state variables as A[1], B[1], ... A[2], B[2], ... A[3], B[3], ...
  This reformulation leads to a banded Jacobian with (upper and lower) half bandwidth = number of species.
  Then the selected integrator solves the banded problem.

• The second method uses lsodes. Based on the dimension of the problem, the method first calculates the sparsity pattern of the Jacobian, under the assumption that transport is only occurring between adjacent layers. Then lsodes is called to solve the problem.
  As lsodes is used to integrate, it may be necessary to specify the length of the real work array, lrw.
Although a reasonable guess of \( \text{lrw} \) is made, it is possible that this will be too low. In this case, \text{ode.1D} \ will return with an error message telling the size of the work array actually needed. In the second try then, set \( \text{lrw} \) equal to this number.

For instance, if you get the error:

```
DLSODES- RWORK length is insufficient to proceed. 
Length needed is .ge. LENRW (=I1), exceeds LRW (=I2) 
In above message, I1 = 27627  I2 = 25932
```

set \( \text{lrv} \) equal to 27627 or a higher value

If the model is specified in compiled code (in a DLL), then option 2, based on \text{lsoedes} is the only solution method.

For single-species 1-D models, you may also use \text{ode.band}.

See the selected integrator for the additional options.

**Value**

A matrix of class \text{deSolve} with up to as many rows as elements in \text{times} and as many columns as elements in \text{y} plus the number of "global" values returned in the second element of the return from \text{func}, plus an additional column (the first) for the time value. There will be one row for each element in \text{times} unless the integrator returns with an unrecoverable error. If \text{y} has a names attribute, it will be used to label the columns of the output value.

The output will have the attributes \text{istate}, and \text{rstate}, two vectors with several useful elements. The first element of \text{istate} returns the conditions under which the last call to the integrator returned. Normal is \text{istate} = 2. If \text{verbose} = \text{TRUE}, the settings of \text{istate} and \text{rstate} will be written to the screen. See the help for the selected integrator for details.

**Note**

It is advisable though not mandatory to specify both \text{nspec} and \text{dimens}. In this case, the solver can check whether the input makes sense (i.e. if \text{nspec} * \text{dimens} == \text{length(y)}).

**Author(s)**

Karline Soetaert <karline.soetaert@nioz.nl>

**See Also**

- \text{ode} for a general interface to most of the ODE solvers,
- \text{ode.band} for integrating models with a banded Jacobian
- \text{ode.2D} for integrating 2-D models
- \text{ode.3D} for integrating 3-D models
- \text{lsoodes, lsoode, lsoada, lsoadar, vode} for the integration options.

\text{diagnostics} to print diagnostic messages.
Examples

```r
lvmod <- function (time, state, parms, N, rr, ri, dr, dri) {
  with (as.list(parms), {
    PREY <- state[,1:N]
    PRED <- state[(N+1):(2*N)]
  }
  }
}
```

```r
# Fluxes due to diffusion
# at internal and external boundaries: zero gradient
FluxPrey <- -Da * diff(c(PREY[1], PREY, PREY[N]))/dri
FluxPred <- -Da * diff(c(PRED[1], PRED, PRED[N]))/dri
```

```r
# Biology: Lotka-Volterra model
Ingestion <- rIng * PREY * PRED
GrowthPrey <- rGrow * PREY * (1-PREY/cap)
MortPredator <- rMort * PRED
```

```r
# Rate of change = Flux gradient + Biology
dPREY <- -diff(ri * FluxPrey)/rr/dr + GrowthPrey - Ingestion
dPRED <- -diff(ri * FluxPred)/rr/dr + Ingestion * assEff - MortPredator
```

```r
return (list(c(dPREY, dPRED)))
}
```

```r
# Model parameters:
R <- 20 # total radius of surface, m
N <- 100 # 100 concentric circles
dr <- R/N # thickness of each layer
r <- seq(dr/2, by = dr, len = N) # distance of center to mid-layer
ri <- seq(0, by = dr, len = N+1) # distance to layer interface
dri <- dr # dispersion distances
```
ode.1D

parms <- c(Da = 0.05, # m2/d, dispersion coefficient
           rIng = 0.2, # /day, rate of ingestion
           rGrow = 1.0, # /day, growth rate of prey
           rMort = 0.2, # /day, mortality rate of pred
           assEff = 0.5, # -, assimilation efficiency
           cap = 10) # density, carrying capacity

## Initial conditions: both present in central circle (box 1) only
state <- rep(0, 2 * N)

## RUNNING the model:
times <- seq(0, 200, by = 1) # output wanted at these time intervals

## the model is solved by the two implemented methods:
## 1. Default: banded reformulation
print(system.time(
  out <- ode.1D(y = state, times = times, func = lvmod, parms = parms,
                nspec = 2, names = c("PREY", "PRED"),
                N = N, rr = r, ri = ri, dr = dr, dri = dri)
))

## 2. Using sparse method
print(system.time(
  out2 <- ode.1D(y = state, times = times, func = lvmod, parms = parms,
                nspec = 2, names = c("PREY","PRED"),
                N = N, rr = r, ri = ri, dr = dr, dri = dri,
                method = "lsodes")
))

## = = = = = = =
## Plotting output
## = = = = = = =
# the data in 'out' consist of: 1st col times, 2-N+1: the prey
# N+2:2*N+1: predators
PREY <- out[, 2:(N + 1)]

filled.contour(x = times, y = r, PREY, color = topo.colors,
               xlab = "time, days", ylab = "Distance, m",
               main = "Prey density")
# similar:
image(out, which = "PREY", grid = r, xlab = "time, days",
      legend = TRUE, ylab = "Distance, m", main = "Prey density")
image(out2, grid = r)

# summaries of 1-D variables
summary(out)

# 1-D plots:
matplot.1D(out, type = "l", subset = time == 10)
matplot.1D(out, type = "l", subset = time > 10 & time < 20)
```r
## Example 2.
## Biochemical Oxygen Demand (BOD) and oxygen (O2) dynamics
## in a river
## #############################################################################

## Model equations
## -------------------
O2BOD <- function(t, state, pars) {
  BOD <- state[1:N]
  O2 <- state[(N+1):(2*N)]

  ## BOD dynamics
  FluxBOD <- v * c(BOD_0, BOD)  # fluxes due to water transport
  FluxO2 <- v * c(O2_0, O2)

  BODrate <- r * BOD              # 1-st order consumption

  # rate of change = flux gradient - consumption + reaeration (O2)
  dBOD <- -diff(FluxBOD)/dx - BODrate
  dO2 <- -diff(FluxO2)/dx - BODrate + p * (O2sat-O2)

  return(list(c(dBOD = dBOD, dO2 = dO2)))
}

## Model application
## -------------------
## parameters
dx <- 25            # grid size of 25 meters
v <- 1e3            # velocity, m/day
x <- seq(dx/2, 5000, by = dx)  # m, distance from river
N <- length(x)
r <- 0.05           # /day, first-order decay of BOD
p <- 0.5            # /day, air-sea exchange rate
O2sat <- 300         # mmol/m3 saturated oxygen conc
O2_0 <- 200         # mmol/m3 riverine oxygen conc
BOD_0 <- 1000       # mmol/m3 riverine BOD concentration

## initial conditions:
state <- c(rep(200, N), rep(200, N))
times <- seq(0, 20, by = 0.1)

## running the model
## step 1 : model spinup
out <- ode.1D(y = state, times, O2BOD, parms = NULL,
             nspec = 2, names = c("BOD", "O2"))

## Plotting output
```
### ode.2D

**Solver for 2-Dimensional Ordinary Differential Equations**

---

#### Description

Solves a system of ordinary differential equations resulting from 2-Dimensional partial differential equations that have been converted to ODEs by numerical differencing.

#### Usage

```r
ode.2D(y, times, func, parms = NULL, nspec = NULL, dimens, method = c("lsodes", "euler", "rk4", "ode23", "ode45", "adams", "iteration"), names = NULL, cyclicBnd = NULL, ...)
```

#### Arguments

- `y`: the initial (state) values for the ODE system, a vector. If `y` has a name attribute, the names will be used to label the output matrix.
- `times`: time sequence for which output is wanted; the first value of `times` must be the initial time.
- `func`: either an \texttt{R}-function that computes the values of the derivatives in the ODE system (the model definition) at time \texttt{t}, or a character string giving the name of a compiled function in a dynamically loaded shared library.
  - If `func` is an \texttt{R}-function, it must be defined as: `func <- function(t, y, parms, ...)`. \texttt{t} is the current time point in the integration, \texttt{y} is the current estimate of the variables in the ODE system. If the initial values \texttt{y} has a names attribute, the names will be available inside \texttt{func}. \texttt{parms} is a vector or list of parameters; \texttt{...} (optional) are any other arguments passed to the function.
  - The return value of `func` should be a list, whose first element is a vector containing the derivatives of \texttt{y} with respect to time, and whose next elements are global values that are required at each point in `times`. The derivatives must be specified in the same order as the state variables \texttt{y}.
- `parms`: parameters passed to `func`.
- `nspec`: the number of species (components) in the model.
ode.2D

**dimens**

2-valued vector with the number of **boxes** in two dimensions in the model.

**cyclicBnd**

if not NULL then a number or a 2-valued vector with the dimensions where a cyclic boundary is used - 1: x-dimension, 2: y-dimension; see details.

**names**

the names of the components; used for plotting.

**method**

the integrator. Use "lsodes" if the model is very stiff; "impAdams" may be best suited for mildly stiff problems; "euler", "rk4", "ode23", "ode45", "adams" are most efficient for non-stiff problems. Also allowed is to pass an integrator function. Use one of the other Runge-Kutta methods via rkMethod. For instance, method = rkMethod("ode45ck") will trigger the Cash-Karp method of order 4(5).

If "lsodes" is used, then also the size of the work array should be specified (lrw) (see lsodes).

Method "iteration" is special in that here the function func should return the new value of the state variables rather than the rate of change. This can be used for individual based models, for difference equations, or in those cases where the integration is performed within func.

... additional arguments passed to lsodes.

**Details**

This is the method of choice for 2-dimensional models, that are only subjected to transport between adjacent layers.

Based on the dimension of the problem, and if lsodes is used as the integrator, the method first calculates the sparsity pattern of the Jacobian, under the assumption that transport is only occurring between adjacent layers. Then lsodes is called to solve the problem.

If the model is not stiff, then it is more efficient to use one of the explicit integration routines.

In some cases, a cyclic boundary condition exists. This is when the first boxes in x-or y-direction interact with the last boxes. In this case, there will be extra non-zero fringes in the Jacobian which need to be taken into account. The occurrence of cyclic boundaries can be toggled on by specifying argument cyclicBnd. For instance, cyclicBnd = 1 indicates that a cyclic boundary is required only for the x-direction, whereas cyclicBnd = c(1,2) imposes a cyclic boundary for both x- and y-direction. The default is no cyclic boundaries.

If lsodes is used to integrate, it will probably be necessary to specify the length of the real work array, lrw.

Although a reasonable guess of lrw is made, it is likely that this will be too low. In this case, ode.2D will return with an error message telling the size of the work array actually needed. In the second try then, set lrw equal to this number.

For instance, if you get the error:

```
DLSODES- RWORK length is insufficient to proceed.
Length needed is .ge. LENRW (=I1), exceeds LRW (=I2)
In above message, I1 = 27627  I2 = 25932
```

set lrw equal to 27627 or a higher value.

See lsodes for the additional options.
Value

A matrix of class deSolve with up to as many rows as elements in times and as many columns as elements in y plus the number of "global" values returned in the second element of the return from func, plus an additional column (the first) for the time value. There will be one row for each element in times unless the integrator returns with an unrecoverable error. If y has a names attribute, it will be used to label the columns of the output value.

The output will have the attributes istate, and rstate, two vectors with several useful elements. The first element of istate returns the conditions under which the last call to the integrator returned. Normal is istate = 2. If verbose = TRUE, the settings of istate and rstate will be written to the screen. See the help for the selected integrator for details.

Note

It is advisable though not mandatory to specify both nspec and dimens. In this case, the solver can check whether the input makes sense (as nspec * dimens[1] * dimens[2] == length(y)).

Do not use this method for problems that are not 2D!

Author(s)

Karline Soetaert <karline.soetaert@nioz.nl>

See Also

- ode for a general interface to most of the ODE solvers,
- ode.band for integrating models with a banded Jacobian
- ode.1D for integrating 1-D models
- ode.3D for integrating 3-D models
- lsodes for the integration options.
diagnostics to print diagnostic messages.

Examples

```r
## A Lotka-Volterra predator-prey model with predator and prey
## dispersing in 2 dimensions
##
## Model definitions
##

lvmod2D <- function (time, state, pars, N, Da, dx) {
  NN <- N*N
  Prey <- matrix(nrow = N, ncol = N, state[1:NN])
  Pred <- matrix(nrow = N, ncol = N, state[(NN+1):(2*NN)])

  with (as.list(pars), {
    ## Biology
```
dPrey <- rGrow * Prey * (1 - Prey/K) - rIng * Prey * Pred
dPred <- rIng * Prey * Pred*assEff - rMort * Pred

zero <- rep(0, N)

## 1. Fluxes in x-direction; zero fluxes near boundaries
FluxPrey <- -Da * rbind(zero,(Prey[2:N,] - Prey[1:(N-1),]), zero)/dx
FluxPred <- -Da * rbind(zero,(Pred[2:N,] - Pred[1:(N-1),]), zero)/dx

## Add flux gradient to rate of change
dPrey <- dPrey - (FluxPrey[2:(N+1),] - FluxPrey[1:N,])/dx
dPred <- dPred - (FluxPred[2:(N+1),] - FluxPred[1:N,])/dx

## 2. Fluxes in y-direction; zero fluxes near boundaries
FluxPrey <- -Da * cbind(zero,(Prey[2:N] - Prey[1:(N-1)]), zero)/dx
FluxPred <- -Da * cbind(zero,(Pred[2:N] - Pred[1:(N-1)]), zero)/dx

## Add flux gradient to rate of change
dPrey <- dPrey - (FluxPrey[2:(N+1)] - FluxPrey[1:N])/dx
dPred <- dPred - (FluxPred[2:(N+1)] - FluxPred[1:N])/dx

return(list(c(as.vector(dPrey), as.vector(dPred))))

# Model applications
# ================

pars <- c(rIng = 0.2, #/day, rate of ingestion
          rGrow = 1.0, #/day, growth rate of prey
          rMort = 0.2, #/day, mortality rate of predator
          assEff = 0.5, #-, assimilation efficiency
          K = 5 ) # mmol/m3, carrying capacity

R <- 20 # total length of surface, m
N <- 50 # number of boxes in one direction
dx <- R/N # thickness of each layer
Da <- 0.05 # m2/d, dispersion coefficient
NN <- N*N # total number of boxes

## initial conditions
yini <- rep(0, 2*N*N)
c <- c(NN/2):(NN/2+1)+N/2, (NN/2):(NN/2+1)-N/2)
yini[cc] <- yini[NN+cc] <- 1

## solve model (5000 state variables... use Cash-Karp Runge-Kutta method
times <- seq(0, 50, by = 1)
out <- ode2D(y = yini, times = times, func = lvmod2D, pars = pars,
            dimens = c(N, N), names = c("Prey", "Pred"),
            N = N, dx = dx, Da = Da, method = rkMethod("rk45ck"))
diagnostics(out)
summary(out)

# Mean of prey concentration at each time step
Prey <- subset(out, select = "Prey", arr = TRUE)
dim(Prey)
MeanPrey <- apply(Prey, MARGIN = 3, FUN = mean)
plot(times, MeanPrey)

## Not run:
## plot results
Col <- colorRampPalette(c("#00007F", "blue", "#007FFF", "cyan",
"#7FFF7F", "yellow", "#FF7F00", "red", "#7F0000"))

for (i in seq(1, length(times), by = 1))
  image(Prey[, , i],
        col = Col[100], xlab = , zlim = range(out[,2:(NN+1)]))

## similar, plotting both and adding a margin text with times:
image(out, xlab = "x", ylab = "y", mtext = paste("time = ", times))

## End(Not run)
select <- c(1, 40)
image(out, xlab = "x", ylab = "y", mtext = "Lotka-Volterra in 2-D",
       subset = select, mfrow = c(2,2), legend = TRUE)

# plot prey and pred at t = 10; first use subset to select data
prey10 <- matrix(nrow = N, ncol = N,
                   data = subset(out, select = "Prey", subset = (time == 10)))
pred10 <- matrix(nrow = N, ncol = N,
                  data = subset(out, select = "Pred", subset = (time == 10)))

mf <- par(mfrow = c(1, 2))
image(prey10)
image(pred10)
par (mfrow = mf)

# same, using deSolve's image:
image(out, subset = (time == 10))

# An example with a cyclic boundary condition,
# Diffusion in 2-D; extra flux on 2 boundaries,
# cyclic boundary in y

diffusion2D <- function(t, Y, par) {
  y <- matrix(nrow = nx, ncol = ny, data = Y) # vector to 2-D matrix
dY <- -r * y # consumption
BNDx <- rep(1, nx)  # boundary concentration
BNDy <- rep(1, ny)  # boundary concentration

## diffusion in X-direction; boundaries=imposed concentration
Flux <- -Dx * rbind(y[1,] - BNDx, (y[2:nx,] - y[1:(nx-1),]), BNDx - y[nx,])/dx
dY2 <- dY - (Flux[2:(nx+1),] - Flux[1:nx])/dx

## diffusion in Y-direction
Flux <- -Dy * cbind(y[1,] - BNDy, (y[2:ny,]-y[1:(ny-1),]), BNDy - y[ny,])/dy
dY1 <- dY - (Flux[2:(ny+1),] - Flux[1:ny])/dy

## extra flux on two sides
dY[,1] <- dY[,1] + 10
dY[,n] <- dY[,n] + 10

## and exchange between sides on y-direction
dY[,ny] <- dY[,1] + (y[,1] - y[,ny]) * 10
return(list(as.vector(dY)))

## parameters
dx <- 1  # grid size
dy <- 1  # diffusion coeff, X- and Y-direction
r <- 0.05  # consumption rate

nx <- 50
ny <- 100
y <- matrix(nrow = nx, ncol = ny, 1)

## model most efficiently solved with lsodes - need to specify lrw
print(system.time(
    ST3 <- ode.2D(y, times = 1:100, func = diffusion2D, parms = NULL,
    dimens = c(nx, ny), verbose = TRUE, names = "y",
    lrw = 400000, atol = 1e-10, rtol = 1e-10, cyclicBnd = 2)
))

# summary of 2-D variable
summary(ST3)

# plot output at t = 10
T10 <- matrix(nrow = nx, ncol = ny,
    data = subset(ST3, select = "y", subset = (time == 10)))
persp(T10, theta = 30, border = NA, phi = 70,
    col = "lightblue", shade = 0.5, box = FALSE)

# image plot, using deSolve's image function
image(ST3, subset = time == 10, method = "persp",
    theta = 30, border = NA, phi = 70, main = "",
    col = "lightblue", shade = 0.5, box = FALSE)

## Not run:
ode.3D

Solver for 3-Dimensional Ordinary Differential Equations

Description

Solves a system of ordinary differential equations resulting from 3-Dimensional partial differential equations that have been converted to ODEs by numerical differencing.

Usage

ode.3D(y, times, func, parms, nspec = NULL, dimens, method = c("lsodes", "euler", "rk4", "ode23", "ode45", "adams", "iteration"), names = NULL, cyclicBnd = NULL, ...)

Arguments

y the initial (state) values for the ODE system, a vector. If y has a name attribute, the names will be used to label the output matrix.

times time sequence for which output is wanted; the first value of times must be the initial time.

func either an R-function that computes the values of the derivatives in the ODE system (the model definition) at time t, or a character string giving the name of a compiled function in a dynamically loaded shared library. If func is an R-function, it must be defined as: func <- function(t, y, parms, ...). t is the current time point in the integration, y is the current estimate of the variables in the ODE system. If the initial values y has a names attribute, the names will be available inside func. parms is a vector or list of parameters; ... (optional) are any other arguments passed to the function.

The return value of func should be a list, whose first element is a vector containing the derivatives of y with respect to time, and whose next elements are global values that are required at each point in times. The derivatives must be specified in the same order as the state variables y.

parms parameters passed to func.

nspec the number of species (components) in the model.
ode.3D

**dimens**
3-valued vector with the number of **boxes** in three dimensions in the model.

**names**
the names of the components; used for plotting.

**cyclicBnd**
if not NULL then a number or a 3-valued vector with the dimensions where a cyclic boundary is used - 1: x-dimension, 2: y-dimension; 3: z-dimension.

**method**
the integrator. Use "lsodes" if the model is very stiff; "impAdams" may be best suited for mildly stiff problems; "euler", "rk4", "ode23", "ode45", "adams" are most efficient for non-stiff problems. Also allowed is to pass an integrator function. Use one of the other Runge-Kutta methods via rkMethod. For instance, method = rkMethod("ode45ck") will trigger the Cash-Karp method of order 4(5).

Method "iteration" is special in that here the function func should return the new value of the state variables rather than the rate of change. This can be used for individual based models, for difference equations, or in those cases where the integration is performed within func.

... additional arguments passed to lsodes.

Details

This is the method of choice for 3-dimensional models, that are only subjected to transport between adjacent layers.

Based on the dimension of the problem, the method first calculates the sparsity pattern of the Jacobian, under the assumption that transport is only occurring between adjacent layers. Then lsodes is called to solve the problem.

As lsodes is used to integrate, it will probably be necessary to specify the length of the real work array, lrw.

Although a reasonable guess of lrw is made, it is likely that this will be too low.

In this case, ode.3D will return with an error message telling the size of the work array actually needed. In the second try then, set lrw equal to this number.

For instance, if you get the error:

```
DLSODES- RWORK length is insufficient to proceed.
Length needed is .ge. LENRW (=I1), exceeds LRW (=I2)
In above message, I1 = 27627 I2 =  25932
```

set lrw equal to 27627 or a higher value.

See lsodes for the additional options.

Value

A matrix of class `deSolve` with up to as many rows as elements in times and as many columns as elements in y plus the number of "global" values returned in the second element of the return from func, plus an additional column (the first) for the time value. There will be one row for each element in times unless the integrator returns with an unrecoverable error. If y has a names attribute, it will be used to label the columns of the output value.

The output will have the attributes istate, and rstate, two vectors with several useful elements. The first element of istate returns the conditions under which the last call to the integrator returned.
Normal is istate = 2. If verbose = TRUE, the settings of istate and rstate will be written to the screen. See the help for the selected integrator for details.

Note

It is advisable though not mandatory to specify both nspec and dimens. In this case, the solver can check whether the input makes sense (as nspec*dimens[1]*dimens[2]*dimens[3] == length(y)). Do not use this method for problems that are not 3D!

Author(s)

Karline Soetaert <karline.soetaert@nioz.nl>

See Also

- `ode` for a general interface to most of the ODE solvers,
- `ode.band` for integrating models with a banded Jacobian
- `ode.1D` for integrating 1-D models
- `ode.2D` for integrating 2-D models
- `lso.des` for the integration options.

See also `diagnostics` to print diagnostic messages.

Examples

```r
## Function to bind two matrices to an array
mbind <- function (Mat1, Array, Mat2, along = 1) {
  dimens <- dim(Array) + c(0, 0, 2)
  if (along == 3)
    array(dim = dimens, data = c(Mat1, Array, Mat2))
  else if (along == 1)
    aperm(array(dim = dimens, data = c(Mat1, aperm(Array, c(3, 2, 1)), Mat2)), c(3, 2, 1))
  else if (along == 2)
    aperm(array(dim = dimens, data = c(Mat1, aperm(Array, c(1, 3, 2)), Mat2)), c(1, 3, 2))
}

yy <- array(dim = c(n, n, n), data = Y) # vector to 3-D array
dY <- -r*yy # consumption
BND <- matrix(nrow = n, ncol = n, data = 1) # boundary concentration

## Diffusion in x-direction
## new array including boundary concentrations in X-direction
BNDx <- -mbind(BND, yy, BND, along = 1)
## Diffusive Flux
```
Flux <- -Dx * (BNDx[2:(n+2),] - BNDx[1:(n+1),])/dx
## rate of change = - flux gradient
dY[] <- dY[] - (Flux[2:(n+1),] - Flux[1:n,])/dx

## diffusion in y-direction
BNDy <- mbind(BND, yy, BND, along = 2)
Flux <- -Dy * (BNDy[,2:(n+2)] - BNDy[,1:(n+1)])/dy
dY[] <- dY[] - (Flux[,2:(n+1)] - Flux[,1:n])/dy

## diffusion in z-direction
BNDz <- mbind(BND, zz, BND, along = 3)
Flux <- -Dz * (BNDz[,2:(n+2)] - BNDz[,1:(n+1)])/dz
dY[] <- dY[] - (Flux[,2:(n+1)] - Flux[,1:n])/dz

return(list(as.vector(dY)))

## parameters
dy <- dx <- dz <- 1     # grid size
Dy <- Dx <- Dz <- 1     # diffusion coeff, X- and Y-direction
r <- 0.025              # consumption rate
n <- 10
y <- array(dim=c(n,n,n),data=10.)

## use lsodes, the default (for n>20, Runge-Kutta more efficient)
print(system.time{
  RES <- ode.3D(y, func = diffusion3D, parms = NULL, dimens = c(n, n, n),
               times = 1:20, lrw = 120000, atol = 1e-10,
               rtol = 1e-10, verbose = TRUE)
})

ty <- array(dim = c(n, n, n), data = RES[nrow(RES), -1])
filled.contour(y[, , n/2], color.palette = terrain.colors)
summary(RES)

## Not run:
for (i in 2:nrow(RES)) {
  y <- array(dim=c(n,n,n),data=RES[i,-1])
  filled.contour(y[,n/2],main=i,color.palette=terrain.colors)
}

## End(Not run)
ode.band

Description

Solves a system of ordinary differential equations.
Assumes a banded Jacobian matrix, but does not rearrange the state variables (in contrast to ode.1D).
Suitable for 1-D models that include transport only between adjacent layers and that model only one
species.

Usage

ode.band(y, times, func, parms, nspec = NULL, dimens = NULL,
bandup = nspec, banddown = nspec, method = "lsode", names = NULL,
...)

Arguments

y the initial (state) values for the ODE system, a vector. If y has a name attribute,
the names will be used to label the output matrix.
times time sequence for which output is wanted; the first value of times must be the
initial time.
func either an R-function that computes the values of the derivatives in the ODE
system (the model definition) at time t, or a character string giving the name of
a compiled function in a dynamically loaded shared library.
If func is an R-function, it must be defined as: func <- function(t, y, parms, ...).
t is the current time point in the integration, y is the current estimate of the vari-
ables in the ODE system. If the initial values y has a names attribute, the names
will be available inside func. parms is a vector or list of parameters; ... (op-
tional) are any other arguments passed to the function.
The return value of func should be a list, whose first element is a vector con-
taining the derivatives of y with respect to time, and whose next elements are
global values that are required at each point in times. The derivatives must be
specified in the same order as the state variables y.
parms parameters passed to func.
nspec the number of *species* (components) in the model.
dimens the number of boxes in the model. If NULL, then nspec should be specified.
bandup the number of nonzero bands above the Jacobian diagonal.
banddown the number of nonzero bands below the Jacobian diagonal.
method the integrator to use, one of "vode", "lsode", "lsoda", "lsodar", "radau".
names the names of the components; used for plotting.
... additional arguments passed to the integrator.

Details

This is the method of choice for single-species 1-D reactive transport models.
For multi-species 1-D models, this method can only be used if the state variables are arranged per
box, per species (e.g. A[1], B[1], A[2], B[2], A[3], B[3], ... for species A, B). By default, the model
function will have the species arranged as A[1], A[2], A[3], ... B[1], B[2], B[3], ... in this case, use ode.1D.

See the selected integrator for the additional options.

Value

A matrix of class desolve with up to as many rows as elements in times and as many columns as elements in y plus the number of "global" values returned in the second element of the return from func, plus an additional column (the first) for the time value. There will be one row for each element in times unless the integrator returns with an unrecoverable error. If y has a names attribute, it will be used to label the columns of the output value.

The output will have the attributes istate and rstate, two vectors with several elements. See the help for the selected integrator for details. the first element of istate returns the conditions under which the last call to the integrator returned. Normal is istate = 2. If verbose = TRUE, the settings of istate and rstate will be written to the screen.

Author(s)

Karline Soetaert <karline.soetaert@nioz.nl>

See Also

• ode for a general interface to most of the ODE solvers,
• ode.1D for integrating 1-D models
• ode.2D for integrating 2-D models
• ode.3D for integrating 3-D models
• lsode, lsoda, lsodar, vode for the integration options.

diagnostics to print diagnostic messages.

Examples

```r
## The Aphid model from Soetaert and Herman, 2009.
## A practical guide to ecological modelling.
## Using R as a simulation platform. Springer.

## 1-D diffusion model

Aphid <- function(t, APHIDS, parameters) {
  deltax <- c(0.5*deltx, rep(deltx, numboxes-1), 0.5*deltx)
  Flux <- -D*diff(c(0, APHIDS, 0))/deltax
  dAPHIDS <- -diff(Flux)/deltax + APHIDS*r
  list(dAPHIDS) # the output
```
## Model application

The model parameters:

- \( D \) <- 0.3  # m²/day diffusion rate
- \( r \) <- 0.01  # /day net growth rate
- \( \text{delx} \) <- 1  # m thickness of boxes
- \( \text{numboxes} \) <- 60

## Distance of boxes on plant, m, 1 m intervals

\( \text{Distance} \) <- seq(from = 0.5, by = \( \text{delx} \), length.out = \( \text{numboxes} \))

## Initial conditions, ind/m²

- \( \text{aphids present only on two central boxes} \)
- \( \text{APHIDS} \) <- rep(0, times = \( \text{numboxes} \))
- \( \text{APHIDS[30:31]} \) <- 1

## State:

\( \text{state} \) <- c(\( \text{APHIDS} = \text{APHIDS} \))  # initialise state variables

## Running the model:

- \( \text{times} \) <- seq(0, 200, by = 1)  # output wanted at these time intervals
- \( \text{out} \) <- ode.band(\( \text{state} \), \( \text{times} \), \( \\text{Aphid} \), \( \text{parms} = 0 \),
  nspec = 1, names = "Aphid")

## Plotting output

```
```

## End

```{r}
```
Description

Plot the output of numeric integration routines.

Usage

```r
## S3 method for class 'deSolve'
plot(x, ..., select = NULL, which = select, ask = NULL,
     obs = NULL, obspar = list(), subset = NULL)

## S3 method for class 'deSolve'
hist(x, select = 1:(ncol(x)-1), which = select, ask = NULL,
     subset = NULL, ...)
## S3 method for class 'deSolve'
image(x, select = NULL, which = select, ask = NULL,
       add.contour = FALSE, grid = NULL,
       method = "image", legend = FALSE, subset = NULL, ...)
## S3 method for class 'deSolve'
subset(x, subset = NULL, select = NULL,
       which = select, arr = FALSE, ...)

plot.1D (x, ..., select = NULL, which = select, ask = NULL,
       obs = NULL, obspar = list(), grid = NULL,
       xyswap = FALSE, delay = 0, vertical = FALSE, subset = NULL)

matplot.0D(x, ..., select = NULL, which = select,
           obs = NULL, obspar = list(), subset = NULL,
           legend = list(x = "topright"))

matplot.1D(x, select = NULL, which = select, ask = NULL,
           obs = NULL, obspar = list(), grid = NULL,
           xyswap = FALSE, vertical = FALSE, subset = NULL, ...)
```

Arguments

- **x**: an object of class deSolve, as returned by the integrators, and to be plotted. For plot.deSolve, it is allowed to pass several objects of class deSolve after x (unnamed) - see second example.
- **which**: the name(s) or the index to the variables that should be plotted or selected. Default = all variables, except time. For use with matplot.0D and matplot.1D, which or select can be a list, with vectors, each referring to a separate y-axis.
- **select**: which variable/columns to be selected. This is added for consistency with the R-function subset.
- **subset**: either a logical expression indicating elements or rows to keep in select, or a vector of integers denoting the indices of the elements over which to loop. Missing values are taken as FALSE
ask logical; if TRUE, the user is asked before each plot, if NULL the user is only asked if more than one page of plots is necessary and the current graphics device is set interactive, see `par(ask)` and `dev.interactive`.

add.contour if TRUE, will add contours to the image plot.

method the name of the plotting method to use, one of "image", "filled.contour", "persp", "contour".

grid only for image plots and for `plot.1D`: the 1-D grid as a vector (for output generated with `ode.1D`), or the x- and y-grid, as a list (for output generated with `ode.2D`).

xyswap if TRUE, then x-and y-values are swapped and the y-axis is from top to bottom. Useful for drawing vertical profiles.

vertical if TRUE, then 1. x-and y-values are swapped, the y-axis is from top to bottom, the x-axis is on top, margin 3 and the main title gets the value of the x-axis. Useful for drawing vertical profiles; see example 2.

delay adds a delay (in milliseconds) between consecutive plots of `plot.1D` to enable animations.

obs a data.frame or matrix with "observed data" that will be added as points to the plots. obs can also be a list with multiple data.frames and/or matrices containing observed data. By default the first column of an observed data set should contain the time-variable. The other columns contain the observed values and they should have names that are known in x.

If the first column of obs consists of factors or characters (strings), then it is assumed that the data are presented in long (database) format, where the first three columns contain (name, time, value).

If obs is not NULL and which is NULL, then the variables, common to both obs and x will be plotted.

obspar additional graphics arguments passed to points, for plotting the observed data. If obs is a list containing multiple observed data sets, then the graphics arguments can be a vector or a list (e.g. for `xlim`, `ylim`), specifying each data set separately.

legend if TRUE, a color legend will be drawn on the right of each image. For use with `matplot.0D` and `matplot.1D`: a list with arguments passed to R-function `legend`.

arr if TRUE, and the output is from a 2-D or 3-D model, an array will be returned with dimension = c(dimension of selected variable, nrow(x)). When `arr=TRUE` then only one variable can be selected. When the output is from a 0-D or 1-D model, then this argument is ignored.

... additional arguments. The graphical arguments are passed to `plot.default`, `image` or `hist` For `plot.desolve`, and `plot.1D`, the dots may contain other objects of class `desolve`, as returned by the integrators, and to be plotted on the same graphs as x - see second example. In this case, x and these other objects should be compatible, i.e. the column names should be the same.

For `plot.desolve`, the arguments after ... must be matched exactly.
Details

The number of panels per page is automatically determined up to 3 x 3 (par(mfrow = c(3, 3))). This default can be overwritten by specifying user-defined settings for mfrow or mfcol. Set mfrow equal to NULL to avoid the plotting function to change user-defined mfrow or mfcol settings.

Other graphical parameters can be passed as well. Parameters are vectorized, either according to the number of plots (xlab, ylab, main, sub, xlim, ylim, log, asp, ann, axes, frame.plot, panel.first, panel.last, cex.lab, cex.axis, cex.main) or according to the number of lines within one plot (other parameters e.g. col, lty, lwd etc.) so it is possible to assign specific axis labels to individual plots, resp. different plotting style. Plotting parameter ylim, or xlim can also be a list to assign different axis limits to individual plots.

Similarly, the graphical parameters for observed data, as passed by obspar can be vectorized, according to the number of observed data sets.

Image plots will only work for 1-D and 2-D variables, as solved with ode.1D and ode.2D. In the first case, an image with times as x- and the grid as y-axis will be created. In the second case, an x-y plot will be created, for all times. Unless ask = FALSE, the user will be asked to confirm page changes. Via argument mtext, it is possible to label each page in case of 2D output.

For images, it is possible to pass an argument method which can take the values "image" (default), “filled.contour”, “contour” or "persp", in order to use the respective plotting method.

plot and matplot.0D will always have times on the x-axis. For problems solved with ode.1D, it may be more useful to use plot.1D or matplot.1D which will plot how spatial variables change with time. These plots will have the grid on the x-axis.

Value

Function subset called with arr = FALSE will return a matrix with up to as many rows as selected by subset and as many columns as selected variables.

When arr = TRUE then an array will be outputted with dimensions equal to the dimension of the selected variable, augmented with the number of rows selected by subset. This means that the last dimension points to times.

Function subset also has an attribute that contains the times selected.

See Also

desolve, ode, print.desolve,

hist image matplot.plot.default for the underlying functions from package graphics,

ode.2D, for an example of using subset with arr = TRUE.

Examples

```r
## Example 1. A Predator-Prey model with 4 species in matrix formulation

LMmatrix <- function(t, n, parms) {
  with(parms, {
    dn <- r * n + n * (A %*% n)
  }
}


```r
return(list(c(dn)))
}
}

parms <- list(
  r = c(r1 = 0.1, r2 = 0.1, r3 = -0.1, r4 = -0.1),
  A = matrix(c(0.0, 0.0, -0.2, 0.01,  # prey 1
               0.0, 0.0, 0.02, -0.1,  # prey 2
               0.2, 0.02, 0.0, 0.0,  # predator 1; prefers prey 1
               0.01, 0.1, 0.0, 0.0),  # predator 2; prefers prey 2
             nrow = 4, ncol = 4, byrow=TRUE)
)

times <- seq(from = 0, to = 500, by = 0.1)
y <- c(prey1 = 1, prey2 = 1, pred1 = 2, pred2 = 2)

out <- ode(y, times, LVmatrix, parms)

## Basic line plot
plot(out, type = "l")

## User-specified axis labels
plot(out, type = "l", ylab = c("Prey 1", "Prey 2", "Pred 1", "Pred 2"),
     xlab = "Time (d)", main = "Time Series")

## Set user-defined mfrow
pm <- par(mfrow = c(2, 2))

## "mfrow=NULL" keeps user-defined mfrow
plot(out, which = c("prey1", "pred2"), mfrow = NULL, type = "l", lwd = 2)

plot(out[,"prey1"], out[,"pred1"], xlab="prey1",
     ylab = "pred1", type = "l", lwd = 2)
plot(out[,"prey2"], out[,"pred2"], xlab = "prey2",
     ylab = "pred2", type = "l", lwd = 2)

## restore graphics parameters
par("mfrow" = pm)

## Plot all in one figure, using matplot
matplot(out, lwd = 2)

## Split y-variables in two groups
matplot(out, which = list(c(1,3), c(2,4)), lty = 1,
        ylab = c("prey1, pred1", "prey2, pred2"))
```

# Example 2. Add second and third output, and observations
#----------------------------------------------------------------------

# New runs with different parameter settings

```r
parms2 <- parms
parms2$r[1] <- 0.2
out2 <- ode(y, times, LVmatrix, parms2)
```
# New runs with different parameter settings
parms3 <- parms
parms3$r[1] <- 0.05
out3 <- ode(y, times, LVmatrix, parms3)

# plot all three outputs
plot(out, out2, out3, type = "l",
ylab = c("Prey 1", "Prey 2", "Pred 1", "Pred 2"),
xlab = "Time (d)", main = "Time Series",
col = c("red", "blue", "darkred"))

## 'observed' data
obs <- as.data.frame(out[out[,1] %in% seq(10, 500, by = 30), ])

plot(out, which = "prey1", type = "l", obs = obs,
     obspar = list(pch = 18, cex = 2))

plot(out, type = "l", obs = obs, col = "red")
matplot.0D(out, which = c("prey1", "pred1"), type = "l", obs = obs)

## second set of 'observed' data and two outputs
obs2 <- as.data.frame(out2[out2[,1] %in% seq(10, 500, by = 50), ])

## manual xlim, log
plot(out, out2, type = "l", obs = list(obs, obs2), col = c("red", "blue"),
     obspar = list(pch = 18:19, cex = 2, col = c("red", "blue")),
     log = c("y", ""), which = c("prey1", "prey1"),
     xlim = list(c(100, 500), c(0, 400)))

## data in 'long' format
OBS <- data.frame(name = c(rep("prey1", 3), rep("prey2", 2)),
                   time = c(10, 100, 250, 10, 400),
                   value = c(0.05, 0.04, 0.7, 0.5, 1))

OBS
plot(out, obs = OBS, obspar = c(pch = 18, cex = 2))

# a subset only:
plot(out, subset = prey1 < 0.5, type = "p")

# Simple histogram
hist(out, col = "darkblue", breaks = 50)

hist(out, col = "darkblue", breaks = 50, subset = prey1<1 & prey2 < 1)

# different parameters per plot
hist(out, col = c("darkblue", "red", "orange", "black"),
     breaks = c(10,50))

## The Aphid model from Soetaert and Herman, 2009.
## A practical guide to ecological modelling.
## Using R as a simulation platform. Springer.

### 1-D diffusion model

### Model equations

Aphid <- function(t, APHIDS, parameters) {
  deltax <- c(0.5*deltax, rep(deltax, numboxes - 1), 0.5*deltax)
  Flux <- -D * diff(c(0, APHIDS, 0))/deltax
  dAPHIDS <- -diff(Flux)/deltax + APHIDS * r
  list(dAPHIDS, Flux = Flux)
}

### Model application

### the model parameters:

D <- 0.3  # m²/day diffusion rate
r <- 0.01 # /day net growth rate
deltax <- 1 # m thickness of boxes
numboxes <- 60

### distance of boxes on plant, m, 1 m intervals
Distance <- seq(from = 0.5, by = deltax, length.out = numboxes)

### Initial conditions, ind/m²

### aphids present only on two central boxes
APHIDS <- rep(0, times = numboxes)
APHIDS[30:31] <- 1

### restricting time

### restricting time

image(out, grid = Distance, main = "Aphid model", ylab = "distance, m", legend = TRUE)

matplot.1D(out, type = "l", lwd = 2, xyswap = TRUE, lty = 1)

matplot.1D(out, type = "l", lwd = 2, xyswap = TRUE, lty = 1,
subset = time < 50

matplot.1D(out, type = "l", lwd = 2, xyswap = TRUE, lty = 1,
subset = time %in% seq(0, 200, by = 10), col = "grey")

## Not run:
plot(out, ask = FALSE, mfrow = c(1, 1))
plot.1D(out, ask = FALSE, type = "l", lwd = 2, xyswap = TRUE)

## End(Not run)

## see help file for ode.2D for images of 2D variables

radau

Implicit Runge-Kutta RADAU IIA

Description

Solves the initial value problem for stiff or nonstiff systems of ordinary differential equations (ODE) in the form:

\[
dy/dt = f(t, y)
\]

or linearly implicit differential algebraic equations in the form:

\[
M dy/dt = f(t, y)
\]

The \texttt{R} function \texttt{radau} provides an interface to the Fortran solver RADAU5, written by Ernst Hairer and G. Wanner, which implements the 3-stage RADAU IIA method. It implements the implicit Runge-Kutta method of order 5 with step size control and continuous output. The system of ODEs or DAEs is written as an \texttt{R} function or can be defined in compiled code that has been dynamically loaded.

Usage

\begin{verbatim}
radau(y, times, func, parms, nind = c(length(y), 0, 0),
   rtol = 1e-6, atol = 1e-6, jacfunc = NULL, jactype = "fullint",
   mass = NULL, massup = NULL, massdown = NULL, rootfunc = NULL,
   verbose = FALSE, nroot = 0, hmax = NULL, hini = 0, ynames = TRUE,
   bandup = NULL, banddown = NULL, maxsteps = 5000,
   dllname = NULL, initfunc = dllname, initpar = parms,
   rpar = NULL, ipar = NULL, nout = 0, outnames = NULL,
   forcings = NULL, initforc = NULL, fcontrol = NULL,
   events=NULL, lags = NULL, ...)
\end{verbatim}
Arguments

\textbf{y} \quad \text{the initial (state) values for the ODE system. If } y \text{ has a name attribute, the names will be used to label the output matrix.}

\textbf{times} \quad \text{time sequence for which output is wanted; the first value of } \text{times must be the initial time; if only one step is to be taken; set } \text{times} = \text{NULL.}

\textbf{func} \quad \text{either an } \text{R}-\text{function that computes the values of the derivatives in the ODE system (the model definition) at time } t, \text{ or the right-hand side of the equation}

\[ M \frac{dy}{dt} = f(t, y) \]

\text{if a DAE. (if } \text{mass is supplied then the problem is assumed a DAE).}

\text{func} \text{ can also be a character string giving the name of a compiled function in a dynamically loaded shared library.}

\text{If } \text{func} \text{ is an } \text{R}-\text{function, it must be defined as:}

\text{func <- function(t, y, parms, ...).}

\text{t} \text{ is the current time point in the integration, } y \text{ is the current estimate of the variables in the ODE system. If the initial values } y \text{ has a names attribute, the names will be available inside } \text{func. parms is a vector or list of parameters; ... (optional) are any other arguments passed to the function.}

\text{The return value of } \text{func} \text{ should be a list, whose first element is a vector containing the derivatives of } y \text{ with respect to time, and whose next elements are global values that are required at each point in } \text{times. The derivatives must be specified in the same order as the state variables } y.

\text{If } \text{func} \text{ is a string, then } \text{dllname must give the name of the shared library (without extension) which must be loaded before } \text{radau(} \text{) is called. See deSolve package vignette "compiledCode" for more details.}

\textbf{parms} \quad \text{vector or list of parameters used in } \text{func or } \text{jacfunc.}

\textbf{nind} \quad \text{if a DAE system: a three-valued vector with the number of variables of index 1, 2, 3 respectively. The equations must be defined such that the index 1 variables precede the index 2 variables which in turn precede the index 3 variables. The sum of the variables of different index should equal } N, \text{ the total number of variables. This has implications on the scaling of the variables, i.e. index 2 variables are scaled by } 1/h, \text{ index 3 variables are scaled by } 1/h^2.

\textbf{rtol} \quad \text{relative error tolerance, either a scalar or an array as long as } y. \text{ See details.}

\textbf{atol} \quad \text{absolute error tolerance, either a scalar or an array as long as } y. \text{ See details.}

\textbf{jacfunc} \quad \text{if not } \text{NULL, an } \text{R} \text{ function that computes the Jacobian of the system of differential equations } \frac{\partial y_i}{\partial y_j}, \text{ or a string giving the name of a function or subroutine in 'dllname' that computes the Jacobian (see vignette "compiledCode" from package deSolve, for more about this option).}

\text{In some circumstances, supplying } \text{jacfunc can speed up the computations, if the system is stiff. The } \text{R calling sequence for } \text{jacfunc is identical to that of } \text{func.}

\text{If the Jacobian is a full matrix, } \text{jacfunc should return a matrix } \frac{\partial y_i}{\partial y_j}, \text{ where the ith row contains the derivative of } \frac{dy_i}{dt} \text{ with respect to } y_j, \text{ or a vector containing the matrix elements by columns (the way } \text{R and FORTRAN store matrices).}

\text{If the Jacobian is banded, } \text{jacfunc should return a matrix containing only the nonzero bands of the Jacobian, rotated row-wise. See example.}
the structure of the Jacobian, one of "fullint", "fullusr", "bandusr" or "bandint" - either full or banded and estimated internally or by user.

the mass matrix. If not NULL, the problem is a linearly implicit DAE and defined as $M \frac{dy}{dt} = f(t, y)$. If the mass-matrix $M$ is full, it should be of dimension $n^2$ where $n$ is the number of $y$-values; if banded the number of rows should be less than $n$, and the mass-matrix is stored diagonal-wise with element $(i, j)$ stored in $mass(i - j + mumas + 1, j)$.

If mass = NULL then the model is an ODE (default)

number of non-zero bands above the diagonal of the mass matrix, in case it is banded.

number of non-zero bands below the diagonal of the mass matrix, in case it is banded.

if not NULL, an R function that computes the function whose root has to be estimated or a string giving the name of a function or subroutine in ‘dllname’ that computes the root function. The R calling sequence for rootfunc is identical to that of func. rootfunc should return a vector with the function values whose root is sought.

if TRUE: full output to the screen, e.g. will print the diagnostics of the integration - see details.

only used if ‘dllname’ is specified: the number of constraint functions whose roots are desired during the integration; if rootfunc is an R-function, the solver estimates the number of roots.

an optional maximum value of the integration stepsize. If not specified, hmax is set to the largest difference in times, to avoid that the simulation possibly ignores short-term events. If 0, no maximal size is specified.

initial step size to be attempted; if 0, the initial step size is set equal to 1e-6. Usually 1e-3 to 1e-5 is good for stiff equations

logical, if FALSE names of state variables are not passed to function func; this may speed up the simulation especially for multi-D models.

number of non-zero bands above the diagonal, in case the Jacobian is banded.

number of non-zero bands below the diagonal, in case the Jacobian is banded.

average maximal number of steps per output interval taken by the solver. This argument is defined such as to ensure compatibility with the Livermore-solvers. RADAU only accepts the maximal number of steps for the entire integration, and this is calculated as length(times) * maxsteps.

a string giving the name of the shared library (without extension) that contains all the compiled function or subroutine definitions refered to in func and jacfunc. See vignette "compiledCode" from package deSolve.

if not NULL, the name of the initialisation function (which initialises values of parameters), as provided in ‘dllname’. See vignette "compiledCode" from package deSolve.

only when ‘dllname’ is specified and an initialisation function initfunc is in the dll: the parameters passed to the initialiser, to initialise the common blocks (FORTRAN) or global variables (C, C++).
**rpar** only when `dllname` is specified: a vector with double precision values passed to the dll-functions whose names are specified by `func` and `jacfunc`.

**ipar** only when `dllname` is specified: a vector with integer values passed to the dll-functions whose names are specified by `func` and `jacfunc`.

**nout** only used if `dllname` is specified and the model is defined in compiled code: the number of output variables calculated in the compiled function `func`, present in the shared library. Note: it is not automatically checked whether this is indeed the number of output variables calculated in the DLL - you have to perform this check in the code. See vignette "compiledCode" from package deSolve.

**outnames** only used if `dllname` is specified and `nout > 0`: the names of output variables calculated in the compiled function `func`, present in the shared library. These names will be used to label the output matrix.

**forcings** only used if `dllname` is specified: a list with the forcing function data sets, each present as a two-columned matrix, with (time, value); interpolation outside the interval [min(times), max(times)] is done by taking the value at the closest data extreme.

See forcings or package vignette "compiledCode".

**initforc** if not NULL, the name of the forcing function initialisation function, as provided in ‘dllname’. It MUST be present if `forcings` has been given a value. See forcings or package vignette "compiledCode".

**fcontrol** A list of control parameters for the forcing functions. See forcings or vignette compiledCode.

**events** A matrix or data frame that specifies events, i.e. when the value of a state variable is suddenly changed. See events for more information.

**lags** A list that specifies timelags, i.e. the number of steps that has to be kept. To be used for delay differential equations. See timelags, dede for more information.

**...** additional arguments passed to `func` and `jacfunc` allowing this to be a generic function.

**Details**

The work is done by the FORTRAN subroutine RADAU5, whose documentation should be consulted for details. The implementation is based on the Fortran 77 version from January 18, 2002.

There are four standard choices for the Jacobian which can be specified with `jactype`.

The options for `jactype` are

- `jactype = "fullint"` a full Jacobian, calculated internally by the solver.
- `jactype = "fullusr"` a full Jacobian, specified by user function `jacfunc`.
- `jactype = "bandusr"` a banded Jacobian, specified by user function `jacfunc`; the size of the bands specified by `bandup` and `banddown`.
- `jactype = "bandint"` a banded Jacobian, calculated by radau; the size of the bands specified by `bandup` and `banddown`. 
Inspection of the example below shows how to specify both a banded and full Jacobian.

The input parameters \( rtol \) and \( atol \) determine the **error control** performed by the solver, which roughly keeps the local error of \( y(i) \) below \( rtol(i) \times abs(y(i)) + atol(i) \).

The diagnostics of the integration can be printed to screen by calling `diagnostics`. If `verbose = TRUE`, the diagnostics will be written to the screen at the end of the integration.

See vignette("deSolve") from the `deSolve` package for an explanation of each element in the vectors containing the diagnostic properties and how to directly access them.

**Models** may be defined in compiled C or FORTRAN code, as well as in an R-function. See package vignette "compiledCode" from package `deSolve` for details.

Information about linking forcing functions to compiled code is in `forcings` (from package `deSolve`).

`radau` can find the root of at least one of a set of constraint functions \( \text{rootfunc} \) of the independent and dependent variables. It then returns the solution at the root if that occurs sooner than the specified stop condition, and otherwise returns the solution according the specified stop condition.

Caution: Because of numerical errors in the function \( \text{rootfun} \) due to roundoff and integration error, `radau` may return false roots, or return the same root at two or more nearly equal values of time.

**Value**

A matrix of class `deSolve` with up to as many rows as elements in `times` and as many columns as elements in `y` plus the number of "global" values returned in the next elements of the return from `func`, plus and additional column for the time value. There will be a row for each element in `times` unless the FORTRAN routine returns with an unrecoverable error. If `y` has a names attribute, it will be used to label the columns of the output value.

**Author(s)**

Karline Soetaert

**References**


**See Also**

- `ode` for a general interface to most of the ODE solvers,
- `ode.1D` for integrating 1-D models,
- `ode.2D` for integrating 2-D models,
- `ode.3D` for integrating 3-D models,
- `daspk` for integrating DAE models up to index 1

`diagnostics` to print diagnostic messages.
Examples

```r
## Example 1: ODE
## Various ways to solve the same model.
##
## the model, 5 state variables
f1 <- function (t, y, parms) {
    ydot <- vector(len = 5)

    return(list(ydot))
}

## the Jacobian, written as a full matrix
fulljac <- function (t, y, parms) {
    jac <- matrix(nrow = 5, ncol = 5, byrow = TRUE,
                  data = c(0.1, -0.2, 0, 0, 0,
                           -0.3, 0.1, -0.2, 0, 0,
                           0, -0.3, 0.1, -0.2, 0,
                           0, 0, -0.3, 0.1, -0.2,
                           0, 0, 0, -0.3, 0.1))
    return(jac)
}

## the Jacobian, written in banded form
bandjac <- function (t, y, parms) {
    jac <- matrix(nrow = 3, ncol = 5, byrow = TRUE,
                  data = c(0, -0.2, -0.2, -0.2, -0.2,
                           0.1, 0.1, 0.1, 0.1, 0.1,
                           -0.3, -0.3, -0.3, -0.3, 0))
    return(jac)
}

## initial conditions and output times
yini <- 1:5
times <- 1:20

## default: stiff method, internally generated, full Jacobian
out <- radau(yini, times, f1, parms = 0)
plot(out)

## stiff method, user-generated full Jacobian
out2 <- radau(yini, times, f1, parms = 0, jactype = "fullusr",
              jacfunc = fulljac)

## stiff method, internally-generated banded Jacobian
```
## one nonzero band above (up) and below(down) the diagonal
out3 <- radau(yini, times, f1, parms = 0, jactype = "bandint",
             bandup = 1, banddown = 1)

## stiff method, user-generated banded Jacobian
out4 <- radau(yini, times, f1, parms = 0, jactype = "banduser",
             jacfunc = bandjac, bandup = 1, banddown = 1)

## Example 2: ODE
## stiff problem from chemical kinetics
Chemistry <- function(t, y, p) {
  dy1 <- -.04*y[1] + 1.0e4*y[2]*y[3]
  dy3 <- 3.0e7*y[2]^2
  list(c(dy1, dy2, dy3))
}
times <- 10^(seq(0, 10, by = 0.1))
yini <- c(y1 = 1.0, y2 = 0, y3 = 0)
out <- radau(func = Chemistry, times = times, y = yini, parms = NULL)
plot(out, log = "x", type = "l", lwd = 2)

## Example 3: DAE
## Car axis problem, index 3 DAE, 8 differential, 2 algebraic equations
## from
## F. Mazzia and C. Magherini. Test Set for Initial Value Problem Solvers,
## release 2.4. Department
## of Mathematics, University of Bari and INdAM, Research Unit of Bari,
## February 2008.
## Available at http://www.dm.uniba.it/~testset.
##
## Problem is written as M*y' = f(t,y,p).
## caraxisfun implements the right-hand side:

caraxisfun <- function(t, y, parms) {
  with(as.list(y), {
    yb <- r * sin(w * t)
    xb <- sqrt(L + L - yb * yb)
    Ll <- sqrt(xl^2 + yl^2)
    Lr <- sqrt((xr - xb)^2 + (yr - yb)^2)
    dxl <- ul; dyl <- vl; dxr <- ur; dyr <- vr
    dul <- (L0-Ll) * xl/Ll + 2 * lam2 * (xl-xr) + lam1*xb
    dv1 <- (L0-Ll) * yl/Ll + 2 * lam2 * (yl-yr) + lam1*yb - k * g
  })
}
rk

Explicit One-Step Solvers for Ordinary Differential Equations (ODE)

Description

Solving initial value problems for non-stiff systems of first-order ordinary differential equations (ODEs).

The R function rk is a top-level function that provides interfaces to a collection of common explicit one-step solvers of the Runge-Kutta family with fixed or variable time steps.

The system of ODE's is written as an R function (which may, of course, use .C, .Fortran, .Call, etc., to call foreign code) or be defined in compiled code that has been dynamically loaded. A vector of parameters is passed to the ODEs, so the solver may be used as part of a modeling package for ODEs, or for parameter estimation using any appropriate modeling tool for non-linear models in R such as optim, nls, nlm or nlme.
Usage

rk(y, times, func, parms, rtol = 1e-6, atol = 1e-6,
verbose = FALSE, tcrit = NULL, hmin = 0, hmax = NULL,
hi = hmax, ynames = TRUE, method = rkMethod("rk45dp7", ... ),
maxsteps = 5000, dllname = NULL, initfunc = dllname,
initpar = parms, rpar = NULL, ipar = NULL,
nout = 0, outnames = NULL, forcings = NULL,
initforc = NULL, fcontrol = NULL, events = NULL, ...)

Arguments

y the initial (state) values for the ODE system. If y has a name attribute, the names will be used to label the output matrix.
times times at which explicit estimates for y are desired. The first value in times must be the initial time.
func either an R-function that computes the values of the derivatives in the ODE system (the model definition) at time t, or a character string giving the name of a compiled function in a dynamically loaded shared library.
If func is an R-function, it must be defined as: func <- function(t, y, parms,...). t is the current time point in the integration, y is the current estimate of the variables in the ODE system. If the initial values y has a names attribute, the names will be available inside func. parms is a vector or list of parameters; ... (optional) are any other arguments passed to the function.
The return value of func should be a list, whose first element is a vector containing the derivatives of y with respect to time, and whose next elements are global values that are required at each point in times. The derivatives must be specified in the same order as the state variables y.
If func is a string, then dllname must give the name of the shared library (without extension) which must be loaded before rk is called. See package vignette "compiledCode" for more details.
parms vector or list of parameters used in func.
rtol relative error tolerance, either a scalar or an array as long as y. Only applicable to methods with variable time step, see details.
atol absolute error tolerance, either a scalar or an array as long as y. Only applicable to methods with variable time step, see details.
tcrit if not NULL, then rk cannot integrate past tcrit. This parameter is for compatibility with other solvers.
verbose a logical value that, when TRUE, triggers more verbose output from the ODE solver.
hmin an optional minimum value of the integration stepsize. In special situations this parameter may speed up computations with the cost of precision. Don’t use hmin if you don’t know why!
hmax an optional maximum value of the integration stepsize. If not specified, hmax is set to the maximum of hini and the largest difference in times, to avoid that the simulation possibly ignores short-term events. If 0, no maximal size is
specified. Note that hmin and hmax are ignored by fixed step methods like "rk4" or "euler".

**hini**
initial step size to be attempted; if 0, the initial step size is determined automatically by solvers with flexible time step. For fixed step methods, setting hini = 0 forces internal time steps identically to external time steps provided by times. Similarly, internal time steps of non-interpolating solvers cannot be bigger than external time steps specified in times.

**ynames**
if FALSE: names of state variables are not passed to function func; this may speed up the simulation especially for large models.

**method**
the integrator to use. This can either be a string constant naming one of the pre-defined methods or a call to function rkMethod specifying a user-defined method. The most common methods are the fixed-step methods "euler", second and fourth-order Runge Kutta ("rk2", "rk4"), or the variable step methods Bogacki-Shampine "rk23bs", Runge-Kutta-Fehlberg "rk34f", the fifth-order Cash-Karp method "rk45ck" or the fifth-order Dormand-Prince method with seven stages "rk45dp7". As a suggestion, one may use "rk23bs" (alias "ode23") for simple problems and "rk45dp7" (alias "ode45") for rough problems.

**maxsteps**
average maximal number of steps per output interval taken by the solver. This argument is defined such as to ensure compatibility with the Livermore-solvers. rk only accepts the maximal number of steps for the entire integration. It is calculated as \( \text{max} = \text{length}(\text{times}) \times \text{maxsteps}, \text{max}(\text{diff}(\text{times})/\text{hini} + 1) \).

**dllname**
a string giving the name of the shared library (without extension) that contains all the compiled function or subroutine definitions refered to in func and jacfunc. See package vignette "compiledCode".

**initfunc**
if not NULL, the name of the initialisation function (which initialises values of parameters), as provided in 'dllname'. See package vignette "compiledCode".

**initpar**
only when 'dllname' is specified and an initialisation function initfunc is in the dll: the parameters passed to the initialiser, to initialise the common blocks (FORTRAN) or global variables (C, C++).

**rpar**
only when 'dllname' is specified: a vector with double precision values passed to the dll-functions whose names are specified by func and jacfunc.

**ipar**
only when 'dllname' is specified: a vector with integer values passed to the dll-functions whose names are specified by func and jacfunc.

**nout**
only used if 'dllname' is specified and the model is defined in compiled code: the number of output variables calculated in the compiled function func, present in the shared library. Note: it is not automatically checked whether this is indeed the number of output variables calculated in the dll - you have to perform this check in the code. See package vignette "compiledCode".

**outnames**
only used if 'dllname' is specified and nout > 0: the names of output variables calculated in the compiled function func, present in the shared library.

**forcings**
only used if 'dllname' is specified: a list with the forcing function data sets, each present as a two-columned matrix, with (time,value); interpolation outside the interval \([\text{min(\text{times})}, \text{max(\text{times})}]\) is done by taking the value at the closest data extreme.

See forcings or package vignette "compiledCode".
initforc if not NULL, the name of the forcing function initialisation function, as provided in 'dllname'. It MUST be present if forcings has been given a value. See forcings or package vignette "compiledCode".

fcontrol A list of control parameters for the forcing functions. See forcings or vignette compiledCode.

events A matrix or data frame that specifies events, i.e. when the value of a state variable is suddenly changed. See events for more information. Not also that if events are specified, then polynomial interpolation is switched off and integration takes place from one external time step to the next, with an internal step size less than or equal the difference of two adjacent points of times.

additional arguments passed to func allowing this to be a generic function.

Details

Function rk is a generalized implementation that can be used to evaluate different solvers of the Runge-Kutta family of explicit ODE solvers. A pre-defined set of common method parameters is in function rkMethod which also allows to supply user-defined Butcher tables.

The input parameters rtol, and atol determine the error control performed by the solver. The solver will control the vector of estimated local errors in y, according to an inequality of the form max-norm of (e/ewt) ≤ 1, where ewt is a vector of positive error weights. The values of rtol and atol should all be non-negative. The form of ewt is:

\[ \text{rtol} \times \text{abs}(y) + \text{atol} \]

where multiplication of two vectors is element-by-element.

Models can be defined in R as a user-supplied R-function, that must be called as: yprime = func(t, y, parms). t is the current time point in the integration, y is the current estimate of the variables in the ODE system.

The return value of func should be a list, whose first element is a vector containing the derivatives of y with respect to time, and whose second element contains output variables that are required at each point in time. Examples are given below.

Value

A matrix of class deSolve with up to as many rows as elements in times and as many columns as elements in y plus the number of "global" values returned in the next elements of the return from func, plus and additional column for the time value. There will be a row for each element in times unless the integration routine returns with an unrecoverable error. If y has a names attribute, it will be used to label the columns of the output value.

Note

Arguments rpar and ipar are provided for compatibility with lsoda.

Starting with version 1.8 implicit Runge-Kutta methods are also supported by this general rk interface, however their implementation is still experimental. Instead of this you may consider radau for a specific full implementation of an implicit Runge-Kutta method.
Author(s)
Thomas Petzoldt <thomas.petzoldt@tu-dresden.de>

References

See Also
For most practical cases, solvers of the Livermore family (i.e. the ODEPACK solvers, see below) are superior. Some of them are also suitable for stiff ODEs, differential algebraic equations (DAEs), or partial differential equations (PDEs).

- `rkMethod` for a list of available Runge-Kutta parameter sets,
- `rk4` and `euler` for special versions without interpolation (and less overhead),
- `lsoda, lsode, lsodes, lsodar, vode, daspk` for solvers of the Livermore family,
- `ode` for a general interface to most of the ODE solvers,
- `ode.band` for solving models with a banded Jacobian,
- `ode.1D` for integrating 1-D models,
- `ode.2D` for integrating 2-D models,
- `ode.3D` for integrating 3-D models,
- `diagnostics` to print diagnostic messages.

Examples
```r
# Example: Resource-producer-consumer Lotka-Volterra model
# Note:
# Parameters are a list, names accessible via "with" function
# Function sigimp passed as an argument (input) to model
# (see also ode and lsoda examples)

SPCMod <- function(t, x, parms, input) {
  with(as.list(c(parms, x)), {
    import <- input(t)
    dS <- import - b*S*P + g*C  # substrate
    dP <- c*S*P - d*C*P         # producer
    dC <- e*P*C - f*C           # consumer
  })
}
```

res <- c(dS, dP, dC)
list(res)
)
)

## The parameters
parms <- c(b = 0.001, c = 0.1, d = 0.1, e = 0.1, f = 0.1, g = 0.0)

## vector of timesteps
times <- seq(0, 200, length = 101)

## external signal with rectangle impulse
signal <- data.frame(times = times,
                      import = rep(0, length(times)))

signal$import[signal$times >= 10 & signal$times <= 11] <- 0.2

sigimp <- approxfun(signal$times, signal$import, rule = 2)

## Start values for steady state
xstart <- c(S = 1, P = 1, C = 1)

## Euler method
out1 <- rk(xstart, times, SPCmod, parms, hini = 0.1,
            input = sigimp, method = "euler")

## classical Runge-Kutta 4th order
out2 <- rk(xstart, times, SPCmod, parms, hini = 1,
            input = sigimp, method = "rk4")

## Dormand-Prince method of order 5(4)
out3 <- rk(xstart, times, SPCmod, parms, hmax = 1,
            input = sigimp, method = "rk45dp7")

mf <- par("mหาย")
## deSolve plot method for comparing scenarios
plot(out1, out2, out3, which = c("S", "P", "C"),
     main = c("Substrate", "Producer", "Consumer"),
     col = c("black", "red", "green"),
     lty = c("solid", "dotted", "dotted"), lwd = c(1, 2, 1))

## user-specified plot function
plot(out1[, "P"], out1[, "C"], type = "l", xlab = "Producer", ylab = "Consumer")
lines(out2[, "P"], out2[, "C"], col = "red", lty = "dotted", lwd = 2)
lines(out3[, "P"], out3[, "C"], col = "green", lty = "dotted")

legend("center", legend = c("euler", "rk4", "rk45dp7"),
        lty = c(1, 3, 3), lwd = c(1, 2, 1),
        col = c("black", "red", "green"))
par(mfrow = mf)
Solve System of ODE (Ordinary Differential Equation)s by Euler’s Method or Classical Runge-Kutta 4th Order Integration.

Description

Solving initial value problems for systems of first-order ordinary differential equations (ODEs) using Euler’s method or the classical Runge-Kutta 4th order integration.

Usage

euler(y, times, func, parms, verbose = FALSE, ynames = TRUE,
    dllname = NULL, initfunc = dllname, initpar = parms,
    rpar = NULL, ipar = NULL, nout = 0, outnames = NULL,
    forcings = NULL, initforc = NULL, fcontrol = NULL, ...)

rk4(y, times, func, parms, verbose = FALSE, ynames = TRUE,
    dllname = NULL, initfunc = dllname, initpar = parms,
    rpar = NULL, ipar = NULL, nout = 0, outnames = NULL,
    forcings = NULL, initforc = NULL, fcontrol = NULL, ...)

euler.1D(y, times, func, parms, nspec = NULL, dimens = NULL,
    names = NULL, verbose = FALSE, ynames = TRUE,
    dllname = NULL, initfunc = dllname, initpar = parms,
    rpar = NULL, ipar = NULL, nout = 0, outnames = NULL,
    forcings = NULL, initforc = NULL, fcontrol = NULL, ...)

Arguments

y the initial (state) values for the ODE system. If y has a name attribute, the names will be used to label the output matrix.

times times at which explicit estimates for y are desired. The first value in times must be the initial time.

func either an R-function that computes the values of the derivatives in the ODE system (the model definition) at time t, or a character string giving the name of a compiled function in a dynamically loaded shared library. If func is an R-function, it must be defined as: func <- function(t, y, parms,...). t is the current time point in the integration, y is the current estimate of the variables in the ODE system. If the initial values y has a names attribute, the names will be available inside func. parms is a vector or list of parameters; ... (optional) are any other arguments passed to the function.

The return value of func should be a list, whose first element is a vector containing the derivatives of y with respect to time, and whose next elements are global values that are required at each point in times. The derivatives must be specified in the same order as the state variables y.
If `func` is a string, then `dllname` must give the name of the shared library (without extension) which must be loaded before `rk4` is called. See package vignette "compiledCode" for more details.

- **parms**: vector or list of parameters used in `func`.
- **nspec**: for 1D models only: the number of *species* (components) in the model. If `NULL`, then `dimens` should be specified.
- **dimens**: for 1D models only: the number of *boxes* in the model. If `NULL`, then `nspec` should be specified.
- **names**: for 1D models only: the names of the components; used for plotting.
- **verbose**: a logical value that, when `true`, triggers more verbose output from the ODE solver.
- **ynames**: if `FALSE`: names of state variables are not passed to function `func`; this may speed up the simulation especially for large models.
- **dllname**: a string giving the name of the shared library (without extension) that contains all the compiled function or subroutine definitions referred to in *func*. See package vignette "compiledCode".
- **initfunc**: if not `NULL`, the name of the initialisation function (which initialises values of parameters), as provided in ‘dllname’. See package vignette "compiledCode".
- **initpar**: only when ‘dllname’ is specified and an initialisation function `initfunc` is in the DLL: the parameters passed to the initialiser, to initialise the common blocks (FORTRAN) or global variables (C, C++).
- **rpar**: only when ‘dllname’ is specified: a vector with double precision values passed to the DLL-functions whose names are specified by `func` and `jacfunc`.
- **ipar**: only when ‘dllname’ is specified: a vector with integer values passed to the dll-functions whose names are specified by `func` and `jacfunc`.
- **nout**: only used if `dllname` is specified and the model is defined in compiled code: the number of output variables calculated in the compiled function `func`, present in the shared library. Note: it is not automatically checked whether this is indeed the number of output variables calculated in the DLL - you have to perform this check in the code. See package vignette "compiledCode".
- **outnames**: only used if ‘dllname’ is specified and `nout > 0`: the names of output variables calculated in the compiled function `func`, present in the shared library.
- **forcings**: only used if ‘dllname’ is specified: a list with the forcing function data sets, each present as a two-columned matrix, with (time, value); interpolation outside the interval \([\min(\text{times}), \max(\text{times})]\) is done by taking the value at the closest data extreme.
  See forcings or package vignette "compiledCode".
- **initforc**: if not `NULL`, the name of the forcing function initialisation function, as provided in ‘dllname’. It MUST be present if forcings has been given a value. See forcings or package vignette "compiledCode".
- **fcontrol**: A list of control parameters for the forcing functions. See forcings or vignette compiledCode.
- **...**: additional arguments passed to `func` allowing this to be a generic function.
Details

rk4 and euler are special versions of the two fixed step solvers with less overhead and less functionality (e.g. no interpolation and no events) compared to the generic Runge-Kutta codes called by ode resp. rk.

If you need different internal and external time steps or want to use events, please use: rk(y, times, func, parms, method = "rk4") or rk(y, times, func, parms, method = "euler").

See help pages of rk and rkMethod for details.

Function euler.1d essentially calls function euler but contains additional code to support plotting of 1D models, see ode.1d and plot.1d for details.

Value

A matrix of class deSolve with up to as many rows as elements in times and as many columns as elements in y plus the number of "global" values returned in the next elements of the return from func, plus and additional column for the time value. There will be a row for each element in times unless the integration routine returns with an unrecoverable error. If y has a names attribute, it will be used to label the columns of the output value.

Note

For most practical cases, solvers with flexible timestep (e.g. rk(method = "ode45"); and especially solvers of the Livermore family (ODEPACK, e.g. lsoda) are superior.

Author(s)

Thomas Petzoldt <thomas.petzoldt@tu-dresden.de>

See Also

- rkMethod for a list of available Runge-Kutta parameter sets,
- rk for the more general Runge-Code,
- lsoda, lsode, lsodes, lsodar, vode, daspk for solvers of the Livermore family,
- ode for a general interface to most of the ODE solvers,
- ode.band for solving models with a banded Jacobian,
- ode.1d for integrating 1-D models,
- ode.2d for integrating 2-D models,
- ode.3d for integrating 3-D models,
- dede for integrating models with delay differential equations,
diagnostics to print diagnostic messages.
Examples

```r
# Example: Analytical and numerical solutions of logistic growth

# the derivative of the logistic
logist <- function(t, x, parms) {
  with(as.list(parms), {
    dx <- r * x[1] * (1 - x[1]/K)
    list(dx)
  })
}

time <- 0:100
N0 <- 0.1; r <- 0.5; K <- 100
parms <- c(r = r, K = K)
x <- c(N = N0)

# analytical solution
plot(time, K/(1 + (K/N0-1) * exp(-r*time)), ylim = c(0, 120),
     type = "l", col = "red", lwd = 2)

# reasonable numerical solution with rk4
out <- as.data.frame(rk4(x, time, logist, parms))
points(out$time, out$n, pch = 16, col = "blue", cex = 0.5)

# same time step with euler, systematic under-estimation
out <- as.data.frame(euler(x, time, logist, parms))
points(out$time, out$n, pch = 1)

# unstable result
out <- as.data.frame(euler(x, time, logist, parms))
points(out$time, out$n, pch = 8, cex = 0.5)

# method with automatic time step
out <- as.data.frame(lsoda(x, time, logist, parms))
points(out$time, out$n, pch = 1, col = "green")
```

---

**rkMethod**

*Collection of Parameter Sets (Butcher Arrays) for the Runge-Kutta Family of ODE Solvers*
rkMethod

Description

This function returns a list specifying coefficients and properties of ODE solver methods from the Runge-Kutta family.

Usage

rkMethod(method = NULL, ...)

Arguments

method a string constant naming one of the pre-defined methods of the Runge-Kutta family of solvers. The most common methods are the fixed-step methods "euler", "rk2", "rk4" or the variable step methods "rk23bs" (alias "ode23"), "rk45dp7" (alias "ode45") or "rk78f".

... specification of a user-defined solver, see Value and example below.

Details

This function supplies method settings for rk or ode. If called without arguments, the names of all currently implemented solvers of the Runge-Kutta family are returned.

The following comparison gives an idea how the algorithms of deSolve are related to similar algorithms of other simulation languages:

<table>
<thead>
<tr>
<th>rkMethod</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;euler&quot;</td>
<td>Euler’s Method</td>
</tr>
<tr>
<td>&quot;rk2&quot;</td>
<td>2nd order Runge-Kutta, fixed time step (Heun’s method)</td>
</tr>
<tr>
<td>&quot;rk4&quot;</td>
<td>classical 4th order Runge-Kutta, fixed time step</td>
</tr>
<tr>
<td>&quot;rk23&quot;</td>
<td>Runge-Kutta, order 2(3); Octave: ode23</td>
</tr>
<tr>
<td>&quot;rk23bs&quot;, &quot;ode23&quot;</td>
<td>Bogacki-Shampine, order 2(3); Matlab: ode23</td>
</tr>
<tr>
<td>&quot;rk34f&quot;</td>
<td>Runge-Kutta-Fehlberg, order 3(4)</td>
</tr>
<tr>
<td>&quot;rk45ck&quot;</td>
<td>Runge-Kutta Cash-Karp, order 4(5)</td>
</tr>
<tr>
<td>&quot;rk45f&quot;</td>
<td>Runge-Kutta-Fehlberg, order 4(5); Octave: ode45, pair=1</td>
</tr>
<tr>
<td>&quot;rk45e&quot;</td>
<td>Runge-Kutta-England, order 4(5)</td>
</tr>
<tr>
<td>&quot;rk45dp6&quot;</td>
<td>Dormand-Prince, order 4(5), local order 6</td>
</tr>
<tr>
<td>&quot;rk45dp7&quot;, &quot;ode45&quot;</td>
<td>Dormand-Prince 4(5), local order 7</td>
</tr>
<tr>
<td>&quot;rk78f&quot;</td>
<td>Runge-Kutta-Fehlberg, order 7(8)</td>
</tr>
<tr>
<td>&quot;rk78dp&quot;</td>
<td>Dormand-Prince, order 7(8)</td>
</tr>
</tbody>
</table>

Note that this table is based on the Runge-Kutta coefficients only, but the algorithms differ also in their implementation, in their stepsize adaption strategy and interpolation methods. The table reflects the state at time of writing and it is of course possible that implementations change.

Methods "rk45dp7" (alias "ode45") and "rk45ck" contain specific and efficient built-in interpolation schemes (dense output).

As an alternative, Neville-Aitken polynomials can be used to interpolate between time steps. This is
available for all RK methods and may be useful to speed up computation if no dense-output formula is available. Note however, that this can introduce considerable local error; it is disabled by default (see nknots below).

**Value**

A list with the following elements:

- **ID**
  - name of the method (character)

- **varstep**
  - boolean value specifying if the method allows for variable time step (TRUE) or not (FALSE).

- **FSAL**
  - (first same as last) optional boolean value specifying if the method allows re-use of the last function evaluation (TRUE) or not (FALSE or NULL).

- **A**
  - coefficient matrix of the method. As link{rk} supports only explicit methods, this matrix must be lower triangular. A must be a vector for fixed step methods where only the subdiagonal values are different from zero.

- **b1**
  - coefficients of the lower order Runge-Kutta pair.

- **b2**
  - coefficients of the higher order Runge-Kutta pair (optional, for embedded methods that allow variable time step).

- **c**
  - coefficients for calculating the intermediate time steps.

- **d**
  - optional coefficients for built-in polynomial interpolation of the outputs from internal steps (dense output), currently only available for method rk45dp7 (Dormand-Prince).

- **densetype**
  - optional integer value specifying the dense output formula; currently only densetype = 1 for rk45dp7 (Dormand-Prince) and densetype = 2 for rk45ck (Cash-Karp) are supported. Undefined values (e.g., densetype = NULL) disable dense output.

- **stage**
  - number of function evaluations needed (corresponds to number of rows in A).

- **qerr**
  - global error order of the method, important for automatic time-step adjustment.

- **nknots**
  - integer value specifying the order of interpolation polynomials for methods without dense output. If nknots < 2 (the default) then internal interpolation is switched off and integration is performed step by step between external time steps.

  If nknots is between 3 and 8, Neville-Aitken polynomials are used, which need at least nknots + 1 internal time steps. Interpolation may speed up integration but can lead to local errors higher than the tolerance, especially if external and internal time steps are very different.

- **alpha**
  - optional tuning parameter for stepsize adjustment. If alpha is omitted, it is set to $1/Qerr - 0.75beta$. The default value is $1/Qerr$ (for beta = 0).

- **beta**
  - optional tuning parameter for stepsize adjustment. Typical values are 0 (default) or $0.4/Qerr$.

**Note**

- Adaptive stepsize Runge-Kuttas are preferred if the solution contains parts when the states change fast, and parts when not much happens. They will take small steps over bumpy ground and long steps over uninteresting terrain.
• As a suggestion, one may use "rk23" (alias "ode23") for simple problems and "rk45dp7" (alias "ode45") for rough problems. The default solver is "rk45dp7" (alias "ode45"), because of its relatively high order (4), re-use of the last intermediate steps (FSAL = first same as last) and built-in polynomial interpolation (dense output).

• Solver "rk23bs", that supports also FSAL, may be useful for slightly stiff systems if demands on precision are relatively low.

• Another good choice, assuring medium accuracy, is the Cash-Karp Runge-Kutta method, "rk45ck".

• Classical "rk4" is traditionally used in cases where an adequate stepsize is known a-priori or if external forcing data are provided for fixed time steps only and frequent interpolation of external data needs to be avoided.

• Method "rk45dp7" (alias "ode45") contains an efficient built-in interpolation scheme (dense output) based on intermediate function evaluations.

Starting with version 1.8 implicit Runge-Kutta (irk) methods are also supported by the general rk interface, however their implementation is still experimental. Instead of this you may consider radau for a specific full implementation of an implicit Runge-Kutta method.

Author(s)

Thomas Petzoldt <thomas.petzoldt@tu-dresden.de>

References


MATLAB (R) is a registed property of The Mathworks Inc. http://www.mathworks.com/
See Also

`rk`, `ode`

Examples

```r
rkMethod()  # returns the names of all available methods
rkmethomd("rk45dp7")  # parameters of the Dormand-Prince 5(4) method
rkmethomd("ode45")  # an alias for the same method

func <- function(t, x, parms) {
  with(as.list(c(parms, x)),{
    dP <- a * P - b * C * P
    dC <- b * P * C - c * C
    res <- c(dP, dC)
    list(res)
  })
}
times <- seq(0, 200, length = 101)
parms <- c(a = 0.1, b = 0.1, c = 0.1)
x <- c(P = 2, C = 1)

## rk using ode45 as the default method
out <- rk(x, times, func, parms)

## all methods can be called also from 'ode' by using rkMethod
out <- ode(x, times, func, parms, method = rkMethod("rk44"))

## 'ode' has aliases for the most common RK methods
out <- ode(x, times, func, parms, method = "ode45")

#==------------------------------------------------------------------------
# Comparison of local error from different interpolation methods
#==------------------------------------------------------------------------

## lsoda with lower tolerances (1e-10) used as reference
o0 <- ode(x, times, func, parms, method = "lsoda", atol = 1e-10, rtol = 1e-10)

## rk45dp7 with hmax = 10 > delta_t = 2
o1 <- ode(x, times, func, parms, method = rkMethod("rk45dp7"), hmax = 10)

## disable dense-output interpolation
## and use only Neville-Aitken polynomials instead
o2 <- ode(x, times, func, parms,
  method = rkMethod("rk45dp7", densetype = NULL, nknots = 5), hmax = 10)

## stop and go: disable interpolation completely
## and integrate explicitly between external time steps
o3 <- ode(x, times, func, parms,
  method = rkMethod("rk45dp7", densetype = NULL, nknots = 0, hmax=10))

## compare different interpolation methods with lsoda
mf <- par("mfnrow" = c(4, 1))
Description

A model that describes oxygen consumption in a marine sediment.

One state variable:

- sedimentary organic carbon,

Organic carbon settles on the sediment surface (forcing function Flux) and decays at a constant rate. The equation is simple:

\[
\frac{dC}{dt} = Flux - kC
\]
This model is written in FORTRAN.

Usage

```r
SCOC(times, y = NULL, parms, Flux, ...)
```

Arguments

times  
time sequence for which output is wanted; the first value of times must be the initial time,
y  
the initial value of the state variable; if NULL it will be estimated based on Flux and parms,
parms  
the model parameter, k,
Flux  
a data set with the organic carbon deposition rates,
...  
any other parameters passed to the integrator ode (which solves the model).

Details

The model is implemented primarily to demonstrate the linking of FORTRAN with R-code.
The source can be found in the ‘doc/examples/dynload’ subdirectory of the package.

Author(s)

Karline Soetaert <karline.soetaert@nioz.nl>

References


See Also

- `ccl4model`, the CCl4 inhalation model.
- `aquaphy`, the algal growth model.

Examples

```r
## Forcing function data
Flux <- matrix(ncol = 2, byrow = TRUE, data = c(1, 0.654, 11, 0.167, 21, 0.060, 41, 0.070, 73, 0.277, 83, 0.186, 93, 0.140, 103, 0.255, 113, 0.231, 123, 0.309, 133, 1.127, 143, 1.923, 153, 1.091, 163, 1.001, 173, 1.691, 183, 1.404, 194, 1.226, 204, 0.767, 214, 0.893, 224, 0.737, 234, 0.772, 244, 0.726, 254, 0.624, 264, 0.439, 274, 0.168, 284, 0.280, 294, 0.202, 304, 0.193, 315, 0.286, 325, 0.599, 335, 1.889, 345, 0.996, 355, 0.681, 365, 1.135))

parms <- c(k = 0.01)
```
timelags

```
times <- 1:365
g <- COC(times, parms = parms, Flux = Flux)
plot(g[, "time"], g[, "Depo"], type = "l", col = "red")
lines(g[, "time"], g[, "Mineralisation"], col = "blue")

## Constant interpolation of forcing function - left side of interval
fcontrol <- list(method = "constant")
g2 <- COC(times, parms = parms, Flux = Flux, fcontrol = fcontrol)
plot(g2[, "time"], g2[, "Depo"], type = "l", col = "red")
lines(g2[, "time"], g2[, "Mineralisation"], col = "blue")
```

### timelags

**Time Lagged Values of State Variables and Derivatives.**

**Description**

Functions lagvalue and lagderiv provide access to past (lagged) values of state variables and derivatives.

They are to be used with function dede, to solve delay differential equations.

**Usage**

```
lagvalue(t, nr)
lagderiv(t, nr)
```

**Arguments**

- **t** the time for which the lagged value is wanted; this should be no larger than the current simulation time and no smaller than the initial simulation time.
- **nr** the number of the lagged value; if NULL then all state variables or derivatives are returned.

**Details**

The lagvalue and lagderiv can only be called during the integration, the lagged time should not be smaller than the initial simulation time, nor should it be larger than the current simulation time. Cubic Hermite interpolation is used to obtain an accurate interpolant at the requested lagged time.

**Value**

a scalar (or vector) with the lagged value(s).

**Author(s)**

Karline Soetaert <karline.soetaert@nioz.nl>
See Also
dede, for how to implement delay differential equations.

Examples

```r
##===========================================================================
## exercise 6 from Shampine and Thompson, 2000
## solving delay differential equations with dde23
##
## two lag values
##===========================================================================

##--------------------------
## the derivative function
##--------------------------
derivs <- function(t, y, parms) {
  History <- function(t) c(cos(t), sin(t))
  if (t < 1)
    lag1 <- History(t - 1)[[1]]
  else
    lag1 <- lagvalue(t - 1)[1] # returns a vector; select first element
  if (t < 2)
    lag2 <- History(t - 2)[[2]]
  else
    lag2 <- lagvalue(t - 2, 2) # faster than lagvalue(t - 2)[2]
  dy1 <- lag1 * lag2
  dy2 <- -y[1] * lag2
  list(c(dy1, dy2), lag1 = lag1, lag2 = lag2)
}

##--------------------------
## parameters
##--------------------------
r <- 3.5; m <- 19

##--------------------------
## initial values and times
##--------------------------
yinit <- c(y1 = 0, y2 = 0)
times <- seq(0, 20, by = 0.01)

##--------------------------
## solve the model
##--------------------------
yout <- dede(y = yinit, times = times, func = derivs,
              parms = NULL, atol = 1e-9)
```
### The predator-prey model with time lags, from Hale
### problem 1 from Shampine and Thompson, 2000
### solving delay differential equations with dde23
###
### a vector with lag values
###

```r
# the derivative function
predprey <- function(t, y, parms) {
  tlag <- t - 1
  if (tlag < 0)
    ylag <- c(80, 30)
  else
    ylag <- lagvalue(tlag) # returns a vector
  list(c(dy1, dy2))
}
```

### parameters
###

```r
a <- 0.25; b <- -0.01; c <- -1; d <- 0.01; m <- 200
```

### initial values and times
###

```r
yinit <- c(y1 = 80, y2 = 30)
times <- seq(0, 100, by = 0.01)
```

### solve the model
###

```r
yout <- dde4(y = yinit, times = times, func = predprey, parms = NULL)
```

### display, plot results
###

```r
plot(yout, type = "l", lwd = 2)
```
Solver for Ordinary Differential Equations (ODE)

Description

Solves the initial value problem for stiff or nonstiff systems of ordinary differential equations (ODE) in the form:
\[ dy/dt = f(t, y) \]

The R function `vode` provides an interface to the FORTRAN ODE solver of the same name, written by Peter N. Brown, Alan C. Hindmarsh and George D. Byrne.

The system of ODE's is written as an R function or be defined in compiled code that has been dynamically loaded.

In contrast to `lsoda`, the user has to specify whether or not the problem is stiff and choose the appropriate solution method.

`vode` is very similar to `lsode`, but uses a variable-coefficient method rather than the fixed-step-interpolate methods in `lsode`. In addition, in `vode` it is possible to choose whether or not a copy of the Jacobian is saved for reuse in the corrector iteration algorithm; In `lsode`, a copy is not kept.

**Usage**

```R
vode(y, times, func, parms, rtol = 1e-6, atol = 1e-6,
     jacfunc = NULL, jactype = "fullint", mf = NULL, verbose = FALSE,
     tcrit = NULL, hmin = 0, hmax = NULL, hini = 0, ynames = TRUE,
     maxord = NULL, bandup = NULL, banddown = NULL, maxsteps = 5000,
     dllname = NULL, initfunc = dllname, initpar = parms, rpar = NULL,
     ipar = NULL, nout = 0, outnames = NULL, forcings=NULL,
     initforc = NULL, fcontrol=NULL, events=NULL, lags = NULL,...)
```

**Arguments**

- `y`: the initial (state) values for the ODE system. If `y` has a name attribute, the names will be used to label the output matrix.
- `times`: time sequence for which output is wanted; the first value of `times` must be the initial time; if only one step is to be taken; set `times = NULL`.
- `func`: either an R-function that computes the values of the derivatives in the ODE system (the model definition) at time `t`, or a character string giving the name of a compiled function in a dynamically loaded shared library.
  If `func` is an R-function, it must be defined as: `func <- function(t, y, parms,...)`. `t` is the current time point in the integration, `y` is the current estimate of the variables in the ODE system. If the initial values `y` has a names attribute, the names will be available inside `func`. `parms` is a vector or list of parameters; ... (optional) are any other arguments passed to the function.
  The return value of `func` should be a list, whose first element is a vector containing the derivatives of `y` with respect to time, and whose next elements are global values that are required at each point in `times`. The derivatives must be specified in the same order as the state variables `y`.
  If `func` is a string, then `dllname` must give the name of the shared library (without extension) which must be loaded before `vode()` is called. See package vignette "compiledCode" for more details.
- `parms`: vector or list of parameters used in `func` or `jacfunc`.
- `rtol`: relative error tolerance, either a scalar or an array as long as `y`. See details.
atol  absolute error tolerance, either a scalar or an array as long as y. See details.

jacfunc  if not NULL, an R function that computes the Jacobian of the system of differential equations \( \frac{\partial y_i}{\partial y_j} \), or a string giving the name of a function or subroutine in ‘dllname’ that computes the Jacobian (see vignette "compiledCode" for more about this option).

In some circumstances, supplying jacfunc can speed up the computations, if the system is stiff. The R calling sequence for jacfunc is identical to that of func.

If the Jacobian is a full matrix, jacfunc should return a matrix \( \frac{\partial y_i}{\partial y_j} \), where the ith row contains the derivative of \( dy_i/dt \) with respect to \( y_j \), or a vector containing the matrix elements by columns (the way R and FORTRAN store matrices). If the Jacobian is banded, jacfunc should return a matrix containing only the nonzero bands of the Jacobian, rotated row-wise. See first example of lsode.

jactype  the structure of the Jacobian, one of "fullint", "fullusr", "bandusr" or "bandint" - either full or banded and estimated internally or by user; overruled if mf is not NULL.

mf  the "method flag" passed to function vode - overrules jactype - provides more options than jactype - see details.

verbose  if TRUE: full output to the screen, e.g. will print the diagnostics of the integration - see details.

tcrit  if not NULL, then vode cannot integrate past tcrit. The FORTRAN routine dvode overshoots its targets (times points in the vector times), and interpolates values for the desired time points. If there is a time beyond which integration should not proceed (perhaps because of a singularity), that should be provided in tcrit.

hmin  an optional minimum value of the integration stepsize. In special situations this parameter may speed up computations with the cost of precision. Don’t use hmin if you don’t know why!

hmax  an optional maximum value of the integration stepsize. If not specified, hmax is set to the largest difference in times, to avoid that the simulation possibly ignores short-term events. If 0, no maximal size is specified.

hini  initial step size to be attempted; if 0, the initial step size is determined by the solver.

ynames  logical; if FALSE: names of state variables are not passed to function func; this may speed up the simulation especially for multi-D models.

maxord  the maximum order to be allowed. NULL uses the default, i.e. order 12 if implicit Adams method (meth = 1), order 5 if BDF method (meth = 2). Reduce maxord to save storage space.

bandup  number of non-zero bands above the diagonal, in case the Jacobian is banded.

banddown  number of non-zero bands below the diagonal, in case the Jacobian is banded.

maxsteps  maximal number of steps per output interval taken by the solver.

dllname  a string giving the name of the shared library (without extension) that contains all the compiled function or subroutine definitions referred to in func and jacfunc. See package vignette "compiledCode".
**initfunc** if not NULL, the name of the initialisation function (which initialises values of parameters), as provided in 'dllname'. See package vignette "compiledCode".

**initpar** only when 'dllname' is specified and an initialisation function initfunc is in the dll: the parameters passed to the initialiser, to initialise the common blocks (FORTRAN) or global variables (C, C++)

**rpar** only when 'dllname' is specified: a vector with double precision values passed to the dll-functions whose names are specified by func and jacfunc.

**ipar** only when 'dllname' is specified: a vector with integer values passed to the dll-functions whose names are specified by func and jacfunc.

**nout** only used if dllname is specified and the model is defined in compiled code: the number of output variables calculated in the compiled function func, present in the shared library. Note: it is not automatically checked whether this is indeed the number of output variables calculated in the dll - you have to perform this check in the code - See package vignette "compiledCode".

**outnames** only used if 'dllname' is specified and nout > 0: the names of output variables calculated in the compiled function func, present in the shared library. These names will be used to label the output matrix.

**forcings** only used if 'dllname' is specified: a list with the forcing function data sets, each present as a two-columned matrix, with (time,value); interpolation outside the interval [min(times), max(times)] is done by taking the value at the closest data extreme.

See forcings or package vignette "compiledCode".

**initforc** if not NULL, the name of the forcing function initialisation function, as provided in 'dllname'. It MUST be present if forcings has been given a value. See forcings or package vignette "compiledCode".

**fcontrol** A list of control parameters for the forcing functions. forcings or package vignette "compiledCode"

**events** A matrix or data frame that specifies events, i.e. when the value of a state variable is suddenly changed. See events for more information.

**lags** A list that specifies timelags, i.e. the number of steps that has to be kept. To be used for delay differential equations. See timelags, dede for more information.

... additional arguments passed to func and jacfunc allowing this to be a generic function.

**Details**

Before using the integrator vode, the user has to decide whether or not the problem is stiff.

If the problem is nonstiff, use method flag mf = 10, which selects a nonstiff (Adams) method, no Jacobian used.

If the problem is stiff, there are four standard choices which can be specified with jac and mf.

The options for jac are

**jac = "fullint"**: a full Jacobian, calculated internally by vode, corresponds to mf = 22,

**jac = "fullusr"**: a full Jacobian, specified by user function jacfunc, corresponds to mf = 21,
**jac = "bandusr"**: a banded Jacobian, specified by user function jacfunc; the size of the bands specified by bandup and banddown, corresponds to $mf = 24$.

**jac = "bandint"**: a banded Jacobian, calculated by vode; the size of the bands specified by bandup and banddown, corresponds to $mf = 25$.

More options are available when specifying $mf$ directly.

The legal values of $mf$ are

10, 11, 12, 13, 14, 15, 20, 21, 22, 23, 24, 25, -11, -12, -14, -15, -21, -22, -24, -25.

$mf$ is a signed two-digit integer, $mf = JSV*(10*METH + MITER)$, where

$JSV = \text{SIGN}(mf)$ indicates the Jacobian-saving strategy: $JSV = 1$ means a copy of the Jacobian is saved for reuse in the corrector iteration algorithm. $JSV = -1$ means a copy of the Jacobian is not saved.

$METH$ indicates the basic linear multistep method: $METH = 1$ means the implicit Adams method. $METH = 2$ means the method based on backward differentiation formulas (BDF-s).

$MITER$ indicates the corrector iteration method: $MITER = 0$ means functional iteration (no Jacobian matrix is involved).

$MITER = 1$ means chord iteration with a user-supplied full (NEQ by NEQ) Jacobian.

$MITER = 2$ means chord iteration with an internally generated (difference quotient) full Jacobian (using NEQ extra calls to func per df/dy value).

$MITER = 3$ means chord iteration with an internally generated diagonal Jacobian approximation (using 1 extra call to func per df/dy evaluation).

$MITER = 4$ means chord iteration with a user-supplied banded Jacobian.

$MITER = 5$ means chord iteration with an internally generated banded Jacobian (using ML+MU+1 extra calls to func per df/dy evaluation).

If $MITER = 1$ or 4, the user must supply a subroutine jacfunc.

The example for integrator *lsode* demonstrates how to specify both a banded and full Jacobian.

The input parameters *rtol*, and *atol* determine the *error control* performed by the solver. If the request for precision exceeds the capabilities of the machine, vode will return an error code. See *lsoda* for details.

The diagnostics of the integration can be printed to screen by calling diagnostics. If *verbose* = TRUE, the diagnostics will be written to the screen at the end of the integration.

See vignette("deSolve") for an explanation of each element in the vectors containing the diagnostic properties and how to directly access them.

**Models** may be defined in compiled C or FORTRAN code, as well as in an R-function. See package vignette "compiledCode" for details.

More information about models defined in compiled code is in the package vignette ("compiledCode"); information about linking forcing functions to compiled code is in forcings.

Examples in both C and FORTRAN are in the ‘dynload’ subdirectory of the deSolve package directory.
Value

A matrix of class `deSolve` with up to as many rows as elements in `times` and as many columns as elements in `y` plus the number of "global" values returned in the next elements of the return from `func`, plus and additional column for the time value. There will be a row for each element in `times` unless the FORTRAN routine ‘vode’ returns with an unrecoverable error. If `y` has a names attribute, it will be used to label the columns of the output value.

Note

From version 1.10.4, the default of `atol` was changed from 1e-8 to 1e-6, to be consistent with the other solvers.

Author(s)

Karline Soetaert <karline.soetaert@nioz.nl>

References

Netlib: [httpZOOwwwNnetlibNorg](http://www.netlib.org)

See Also

- `rk`,
- `rk4` and `euler` for Runge-Kutta integrators.
- `lsoda`, `lsode`, `lsodes`, `lsodar`, `daspk` for other solvers of the Livermore family,
- `ode` for a general interface to most of the ODE solvers,
- `ode.band` for solving models with a banded Jacobian,
- `ode.1D` for integrating 1-D models,
- `ode.2D` for integrating 2-D models,
- `ode.3D` for integrating 3-D models,
- `diagnostics` to print diagnostic messages.
Examples

```r
## ex. 1
## The famous Lorenz equations: chaos in the earth's atmosphere
##
chaos <- function(t, state, parameters) {
  with(as.list(c(state)), {
    dx <- -8/3 * x + y * z
    dy <- -10 * (y - z)
    dz <- -x * y + 28 * y - z
    list(c(dx, dy, dz))
  })
}

state <- c(x = 1, y = 1, z = 1)
times <- seq(0, 100, 0.01)

out <- vode(state, times, chaos, 0)

plot(out, type = "l")  # all versus time
plot(out[,"x"], out[,"y"], type = "l", main = "Lorenz butterfly",
     xlab = "x", ylab = "y")

## ex. 2
## SCOC model, in FORTRAN - to see the FORTRAN code:
## browseURL(paste(system.file(package="deSolve"),
##         "doc/examples/dynload/scoc.f",sep=""))
## example from Soetaert and Herman, 2009, chapter 3. (simplified)
##
## Forcing function data
Flux <- matrix(ncol = 2, byrow = TRUE, data = c(
  1, 0.654, 11, 0.167, 21, 0.060, 41, 0.070, 73, 0.277, 83, 0.186,
  93, 0.140,103, 0.255,113, 0.231,123, 0.309,133, 1.127,143, 1.923,
  153,1.091,163, 1.001,173, 1.691,183, 1.404,194, 1.226,204, 0.767,
  214,0.893,224, 0.737,234, 0.772,244, 0.726,254, 0.624,264, 0.439,
  274,0.168,284, 0.280,294, 0.202,304, 0.193,315, 0.286,325, 0.599,
  335,1.889,345, 0.996,355, 0.681,365, 1.135))

parms <- c(k = 0.01)

meanDepo <- mean(approx(Flux[,1], Flux[,2], xout = seq(1, 365, by = 1))$y)

Yini <- c(y = as.double(meanDepo/parms))
times <- 1:365
```
zvode

Solver for Ordinary Differential Equations (ODE) for COMPLEX variables

Description

Solves the initial value problem for stiff or nonstiff systems of ordinary differential equations (ODE) in the form:

\[ \frac{dy}{dt} = f(t, y) \]

where \( dy \) and \( y \) are complex variables.

The \( R \) function \texttt{zvode} provides an interface to the FORTRAN ODE solver of the same name, written by Peter N. Brown, Alan C. Hindmarsh and George D. Byrne.
Usage

zvode(y, times, func, parms, rtol = 1e-6, atol = 1e-6,
     jacfunc = NULL, jactype = "fullint", mf = NULL, verbose = FALSE,
     tcrit = NULL, hmin = 0, hmax = NULL, hini = 0, ynames = TRUE,
     maxord = NULL, bandup = NULL, banddown = NULL, maxsteps = 5000,
     dllname = NULL, initfunc = dllname, initpar = parms, rpar = NULL,
     ipar = NULL, nout = 0, outnames = NULL, forcings = NULL,
     initforc = NULL, fcontrol = NULL, ...)}

Arguments

y
the initial (state) values for the ODE system. If y has a name attribute, the names
will be used to label the output matrix. y has to be complex

times
time sequence for which output is wanted; the first value of times must be the
initial time; if only one step is to be taken; set times = NULL.

func
either an R-function that computes the values of the derivatives in the ODE
system (the model definition) at time t, or a character string giving the name of
a compiled function in a dynamically loaded shared library.
If func is an R-function, it must be defined as: func <- function(t, y, parms, ...).
t is the current time point in the integration, y is the current estimate of the vari-
able in the ODE system. If the initial values y has a names attribute, the names
will be available inside func. parms is a vector or list of parameters; ... (op-
tional) are any other arguments passed to the function.
The return value of func should be a list, whose first element is a vector con-
taining the derivatives of y with respect to time, and whose next elements are
global values that are required at each point in times. The derivatives must be
specified in the same order as the state variables y. They should be complex
numbers.
If func is a string, then dllname must give the name of the shared library (with-
out extension) which must be loaded before zvode() is called. See package
vignette "compiledCode" for more details.

parms
vector or list of parameters used in func or jacfunc.

rtol
relative error tolerance, either a scalar or an array as long as y. See details.
atol
absolute error tolerance, either a scalar or an array as long as y. See details.
jacfunc
if not NULL, an R function that computes the Jacobian of the system of differen-
tial equations ∂y_i/∂y_j, or a string giving the name of a function or subroutine in
'dllname' that computes the Jacobian (see vignette "compiledCode" for more
about this option).
In some circumstances, supplying jacfunc can speed up the computations, if
the system is stiff. The R calling sequence for jacfunc is identical to that of
func.
If the Jacobian is a full matrix, jacfunc should return a matrix dy/dy, where the
ith row contains the derivative of dy_i/dt with respect to y_j, or a vector contain-
ing the matrix elements by columns (the way R and FORTRAN store matrices).
Its elements should be complex numbers.
If the Jacobian is banded, \texttt{jacfunc} should return a matrix containing only the nonzero bands of the Jacobian, rotated row-wise. See first example of \texttt{lsode}.

\textbf{jactype} \\
the structure of the Jacobian, one of "fullint", "fullusr", "bandusr" or "bandint" - either full or banded and estimated internally or by user; overruled if \texttt{mf} is not NULL.

\textbf{mf} \\
the "method flag" passed to function \texttt{zvode} - overrules \texttt{jactype} - provides more options than \texttt{jactype} - see details.

\textbf{verbose} \\
if TRUE: full output to the screen, e.g. will print the diagnostiscs of the integration - see details.

\textbf{tcrit} \\
if not NULL, then \texttt{zvode} cannot integrate past \texttt{tcrit}. The FORTRAN routine \texttt{dvode} overshoots its targets (times points in the vector \texttt{times}), and interpolates values for the desired time points. If there is a time beyond which integration should not proceed (perhaps because of a singularity), that should be provided in \texttt{tcrit}.

\textbf{hmin} \\
an optional minimum value of the integration stepsize. In special situations this parameter may speed up computations with the cost of precision. Don’t use \texttt{hmin} if you don’t know why!

\textbf{hmax} \\
an optional maximum value of the integration stepsize. If not specified, \texttt{hmax} is set to the largest difference in \texttt{times}, to avoid that the simulation possibly ignores short-term events. If 0, no maximal size is specified.

\textbf{hini} \\
initial step size to be attempted; if 0, the initial step size is determined by the solver.

\textbf{ynames} \\
logical; if FALSE: names of state variables are not passed to function \texttt{func}; this may speed up the simulation especially for multi-D models.

\textbf{maxord} \\
the maximum order to be allowed. NULL uses the default, i.e. order 12 if implicit Adams method (\texttt{meth = 1}), order 5 if BDF method (\texttt{meth = 2}). Reduce \texttt{maxord} to save storage space.

\textbf{bandup} \\
number of non-zero bands above the diagonal, in case the Jacobian is banded.

\textbf{banddown} \\
number of non-zero bands below the diagonal, in case the Jacobian is banded.

\textbf{maxsteps} \\
maximum number of steps per output interval taken by the solver.

\textbf{dllname} \\
a string giving the name of the shared library (without extension) that contains all the compiled function or subroutine definitions referred to in \texttt{func} and \texttt{jacfunc}. See package vignette "compiledCode".

\textbf{initfunc} \\
if not NULL, the name of the initialisation function (which initialises values of parameters), as provided in ‘dllname’. See package vignette "compiledCode".

\textbf{initpar} \\
only when ‘\texttt{dllname}’ is specified and an initialisation function \texttt{initfunc} is in the dll: the parameters passed to the initialiser, to initialise the common blocks (FORTRAN) or global variables (C, C++).

\textbf{rpar} \\
only when ‘\texttt{dllname}’ is specified: a vector with double precision values passed to the DLL-functions whose names are specified by \texttt{func} and \texttt{jacfunc}.

\textbf{ipar} \\
only when ‘\texttt{dllname}’ is specified: a vector with integer values passed to the dll-functions whose names are specified by \texttt{func} and \texttt{jacfunc}. 
nout only used if dllname is specified and the model is defined in compiled code: the number of output variables calculated in the compiled function func, present in the shared library. Note: it is not automatically checked whether this is indeed the number of output variables calculated in the DLL - you have to perform this check in the code - See package vignette "compiledCode".

outnames only used if ‘dllname’ is specified and nout > 0: the names of output variables calculated in the compiled function func, present in the shared library. These names will be used to label the output matrix.

forcings only used if ‘dllname’ is specified: a list with the forcing function data sets, each present as a two-columned matrix, with (time,value); interpolation outside the interval [min(times), max(times)] is done by taking the value at the closest data extreme. See forcings or package vignette "compiledCode".

initforc if not NULL, the name of the forcing function initialisation function, as provided in ‘dllname’. It MUST be present if forcings has been given a value. See forcings or package vignette "compiledCode".

fcontrol A list of control parameters for the forcing functions. forcings or package vignette "compiledCode"

... additional arguments passed to func and jacfunc allowing this to be a generic function.

Details

see vode, the double precision version, for details.

Value

A matrix of class deSolve with up to as many rows as elements in times and as many columns as elements in y plus the number of "global" values returned in the next elements of the return from func, plus and additional column for the time value. There will be a row for each element in times unless the FORTRAN routine ‘zvode’ returns with an unrecoverable error. If y has a names attribute, it will be used to label the columns of the output value.

Note

From version 1.10.4, the default of atol was changed from 1e-8 to 1e-6, to be consistent with the other solvers.

The following text is adapted from the zvode.f source code:

When using zvode for a stiff system, it should only be used for the case in which the function f is analytic, that is, when each f(i) is an analytic function of each y(j). Analyticity means that the partial derivative df(i)/dy(j) is a unique complex number, and this fact is critical in the way zvode solves the dense or banded linear systems that arise in the stiff case. For a complex stiff ODE system in which f is not analytic, zvode is likely to have convergence failures, and for this problem one should instead use ode on the equivalent real system (in the real and imaginary parts of y).

Author(s)

Karline Soetaert <karline.soetaert@nioz.nl>
References


Netlib: http://www.netlib.org

See Also

vode for the double precision version

Examples

```r
## Example 1 - very simple example
## df/dt = 1i*f, where 1i is the imaginary unit
## The initial value is f(0) = 1 = 1+0i
##
ZODE <- function(Time, f, Pars) {
  df <- 1i*f
  return(list(df))
}

pars <- NULL
yini <- c(f = 1+0i)
times <- seq(0, 2*pi, length = 100)
out <- zvode(func = ZODE, y = yini, pars = pars, times = times,
             atol = 1e-10, rtol = 1e-10)

# The analytical solution to this ODE is the exp-function:
# f(t) = exp(1i*t)
#       = cos(t)+1i*sin(t) (due to Euler's equation)

analytical_solution <- exp(1i * times)

## compare numerical and analytical solution
tail(cbind(out[,2], analytical_solution))
```
```
## Example 2 - example in "zvode.f",
## df/dt = 1i*f          (same as above ODE)
## dg/dt = -1i*g*f       (an additional ODE depending on f)
##
## Initial values are
## g(0) = 1/2.1 and
## z(0) = 1

ZODE2<-function(Time,State,Pars) {
  with(as.list(State), {
    df <- 1i * f
    dg <- -1i * g * g * f
    return(list(c(df, dg)))
  })
}

yini <- c(f = 1 + 0i, g = 1/2.1 + 0i)
times <- seq(0, 2*pi, length = 100)
out <- zvode(func = ZODE2, y = yini, parms = NULL, times = times,
             atol = 1e-10, rtol = 1e-10)

## The analytical solution is
## f(t) = exp(1i*t)      (same as above)
## g(t) = 1/(f(t) + 1.1)

analytical <- cbind(f = exp(1i * times), g = 1/(exp(1i * times) + 1.1))

## compare numerical solution and the two analytical ones:
tail(cbind(out[,]2, analytical[,]1))
```
Index

*Topic datasets
  ccl4data, 8
*Topic hplot
  plot.deShow, 95
*Topic math
  daspk, 13
  lsoda, 42
  lsodar, 49
  lsode, 55
  lsodes, 63
  ode, 71
  ode.1D, 77
  ode.2D, 83
  ode.3D, 89
  ode.band, 92
  radau, 102
  rk, 109
  rk4, 115
  rkMethod, 119
  vode, 128
  zvode, 135
*Topic misc
  cleanEventTimes, 12
*Topic models
  aquaphy, 5
  ccl4model, 9
  SCOC, 123
*Topic package
  deShow-package, 2
*Topic utilities
  deShow, 23
  diagnostics, 27
  diagnostics.deShow, 28
  DLLfunc, 29
  DLLres, 31
  events, 33
  forcings, 39
  timeLags, 125
  .C, 13, 42, 109
  .Call, 13, 42, 109
  .Fortran, 42, 109
  approx, 40
  approxfun, 39, 40
  aquaphy, 5, 10, 73, 124
  ccl4data, 8, 10
  ccl4model, 6, 9, 35, 73, 124
  cleanEventTimes, 12, 33, 34
  daspk, 4, 13, 32, 46, 53, 60, 68, 73, 106, 113, 117, 133
dede, 4, 17, 23, 45, 52, 58, 66, 73, 105, 117, 126, 131
desolve, 98
desolve (desolve-package), 2
desolve-package, 2
dev.interactive, 97
diagnostics, 18, 19, 27, 45, 46, 53, 54, 59, 60, 67, 68, 73, 79, 85, 91, 94, 106, 113, 117, 132, 133
diagnostics.desolve, 28, 28
DLLfunc, 4, 29
DLLres, 4, 31
euler, 4, 19, 46, 53, 60, 68, 113, 133
euler (rk4), 115
events, 4, 13, 17, 33, 40, 45, 49, 52, 56, 58, 66, 73, 105, 112, 131
fme, 3
gnls, 46
hist, 97, 98
hist.desolve (plot.desolve), 95
image, 97, 98
image.deSolve(plot.deSolve), 95
lagderiv, 4, 24
lagderiv(times), 125
lagvalue, 4, 24
lagvalue(times), 125
legend, 97
lsoda, 4, 18, 19, 24, 34, 42, 49, 52, 53, 56, 59,
60, 67, 68, 72, 73, 79, 94, 113, 117, 129, 132, 133
lsodar, 4, 19, 35, 46, 49, 60, 68, 73, 79, 94,
113, 117, 133
lsode, 4, 19, 34, 43, 46, 50, 53, 55, 57, 68, 73,
79, 94, 113, 117, 129, 130, 132, 133
lsodes, 4, 19, 34, 46, 53, 60, 63, 73, 79, 84,
85, 90, 91, 113, 117, 133
matplot, 98
matplot.0D(plot.deSolve), 95
matplot.1D(plot.deSolve), 95
matplot.deSolve(plot.deSolve), 95
nearestEvent(cleanEventTimes), 12
nlm, 3, 42, 109
nlme, 3, 42, 109
nls, 3, 42, 109
ode, 4, 19, 30, 46, 53, 60, 68, 71, 79, 85, 91,
94, 98, 106, 113, 117, 119, 122, 133
ode.1D, 4, 19, 46, 53, 60, 68, 73, 77, 85, 91,
94, 98, 106, 113, 117, 133
ode.2D, 4, 19, 46, 53, 60, 68, 73, 79, 83, 91,
94, 98, 106, 113, 117, 133
ode.3D, 4, 19, 46, 53, 60, 68, 73, 79, 85, 89,
94, 106, 113, 117, 133
ode.band, 4, 19, 46, 53, 60, 68, 73, 79, 85, 91,
92, 113, 117, 133
optim, 3, 42, 109
par, 97
plot, 98
plot.1D, 117
plot.1D(plot.deSolve), 95
plot.default, 97
plot.deSolve, 73, 95
print.deSolve, 98
print.deSolve(ode), 71
rk4, 4, 19, 46, 53, 60, 68, 73, 109, 117, 119,
122, 133
rkMethod, 4, 19, 46, 53, 72, 73, 111–113, 117, 118
roots, 53
roots(events), 33
SCOC, 123
subset.deSolve(plot.deSolve), 95
summary.deSolve(ode), 71
timelags, 17, 45, 52, 58, 66, 105, 125, 131
vode, 4, 19, 46, 53, 56, 60, 68, 73, 79, 94, 113,
117, 128, 138, 139
zvode, 135